

# Consistentie en beschikbaarheidstesten voor gedistribueerde database systemen

Thomas Uyttendaele

Thesis voorgedragen tot het behalen  
van de graad van Master of Science  
in de ingenieurswetenschappen:  
computerwetenschappen,  
hoofdspecialisatie Gedistribueerde  
systemen

**Promotor:**  
Prof. dr. ir. Wouter Joosen

**Assessor:**  
Prof. dr. ir. Tias Guns,  
Prof. dr. ir. Christophe Huygens

**Begeleider:**  
Dr. ir. Bart Vanbrabant  
Dr. Bert Lagaisse

© Copyright KU Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor als de auteur is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 of via e-mail [info@cs.kuleuven.be](mailto:info@cs.kuleuven.be).

Voorafgaande schriftelijke toestemming van de promotor is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

# Voorwoord

Het afgelopen jaar heeft mijn thesis mij talrijke nieuwe ervaringen en uitdagingen gegeven. Het heeft mij kennis bijgebracht over de verschillende database systemen en de probleemwereld van het installeren, configureren, onderhouden en het gedrag van data systemen.

Deze thesis is tot stand gekomen door de hulp en steun van verschillende mensen, ik neem graag even de tijd om deze te bedanken.

Ik zou graag beginnen met mijn begeleiders Bart Vanbrabant en Bert Lagaisse voor hun permanente begeleiding en hulp bij het onderzoeken en schrijven van de thesis, klaar om met een fris oog en ander perspectief naar het werk te kijken.

Mijn promotor Wouter Joosen zou ik graag willen bedanken voor het vertrouwen en de ondersteuning doorheen het jaar.

Met mijn studiegenoot Arnaud Schoonjans heb ik gesprekken gehad over de mogelijke testmethodes, een grotere variëteit aan database systemen kunnen zien, ook hiervoor bedankt.

Tenslotte wil ik mijn familie, studiegenoten en vrienden bedanken voor hun steun in de thesis en het brengen van extra motivatie op momenten op de momenten dat ik het nodig vond.

*Thomas Uyttendaele*

# Inhoudsopgave

<b>Voorwoord</b>	<b>i</b>
<b>Samenvatting</b>	<b>iv</b>
<b>1 Inleiding</b>	<b>1</b>
1.1 Relationale en NoSQL DBMS's . . . . .	1
1.2 Vergelijking van DBMS's naar performantie en CAP . . . . .	6
1.3 Doelstelling en bijdrage . . . . .	9
1.4 Conclusie . . . . .	10
<b>2 Voorgestelde testmethodiek</b>	<b>11</b>
2.1 Stap 1: Opstellen van de testomgeving . . . . .	12
2.2 Stap 2: Calibratie van de testomgeving . . . . .	12
2.3 Stap 3: Testen van de systemen . . . . .	14
2.4 Stap 4: Verzamelen en analyseren van de testdata . . . . .	18
2.5 Conclusie . . . . .	18
<b>3 Implementatie</b>	<b>21</b>
3.1 Selectie van de DBMS's . . . . .	21
3.2 Gedetailleerde besprekking van de model DBMS's . . . . .	22
3.3 Selectie en uitwerking van de testsoftware . . . . .	29
3.4 Installatie en opstelling van de DBMS's en YCSB . . . . .	30
3.5 Uitvoeren van de calibratie en testen . . . . .	31
3.6 Verzamelen en analyse van de testresultaten . . . . .	33
3.7 Conclusie . . . . .	34
<b>4 Observaties</b>	<b>37</b>
4.1 Calibratie . . . . .	37
4.2 Beschikbaarheidstest . . . . .	42
4.3 Consistentietest . . . . .	46
4.4 Conclusie . . . . .	54
<b>5 Analyse van de resultaten</b>	<b>55</b>
5.1 Calibratie . . . . .	55
5.2 Beschikbaarheidstest . . . . .	55
5.3 Consistentietest . . . . .	58
5.4 Conclusie . . . . .	62
<b>6 Conclusie</b>	<b>63</b>

6.1	Verder werk . . . . .	64
<b>A</b>	<b>Besprekking van verschillende DBMS's</b>	<b>67</b>
A.1	Column database . . . . .	67
A.2	Document database . . . . .	68
A.3	Key-Value database . . . . .	69
A.4	Relationele database . . . . .	71
<b>B</b>	<b>Overzicht van gedetailleerde implementatie keuzes</b>	<b>73</b>
<b>C</b>	<b>Figuren van de observaties</b>	<b>77</b>
<b>D</b>	<b>Opstellen van de testomgeving en uitvoering van de testen</b>	<b>79</b>
D.1	Uitwerking in IMP . . . . .	79
D.2	Uitvoeren testen . . . . .	88
<b>E</b>	<b>Paper</b>	<b>91</b>
<b>F</b>	<b>Poster</b>	<b>97</b>
	<b>Bibliografie</b>	<b>99</b>

# Samenvatting

De meest gebruikte database management systemen zijn momenteel de relationele of NoSQL systemen waarbij er verschillen zijn naar het ondersteunde datamodel. Daarnaast zijn er ook verschillende keuze gemaakt naar het opzetten van deze systemen in een gedistribueerde omgeving.

Volgens het CAP theorema, kan een systeem niet tegelijk consistentie en hoge beschikbaarheid aanbieden en partitie tolerant te zijn. Deze thesis introduceert een model om verschillende database systemen gekwantificeerd te vergelijken naar consistentie en beschikbaarheid. Bij de consistentie testen wordt het leesgedrag vergeleken bij het invoegen of aanpassen van data: hoelang het duurt vooraleer de data beschikbaar is voor elke gebruiker en het gedrag van de query gedurende deze periode.

Voor de beschikbaarheidstesten wordt het gedrag van de queries opgevolgd terwijl een database instantie stopt en of er netwerk partities ontstaan. Dit model is praktisch uitgewerkt met behulp van YCSB en verschillende database systemen zijn getest.

Waar zowel HBase als MongoDB stikte consistentie afleveren, is er een praktisch verschil in hun gedrag: bij HBase worden leesbewerkingen, gestart na de schrijfbewerking, uitgesteld om de nieuwe data terug te geven onmiddellijk na het voltooien van de schrijfbewerking. Bij MongoDB daarin tegen kunnen leesbewerkingen de nieuwe data al teruggeven vooraleer de schrijfbewerking is voltooid. Ook zijn er verschillend garanties bij MongoDB mogelijk voor de lees- en schrijfbewerkingen.

Bij de beschikbaarheidstesten zijn er verschillen tussen de aanpak bij MongoDB, HBase en Pgpool-II. Bij MongoDB en HBase worden er sessies gebruikt en kan de data enige tijd onbeschikbaar zijn bij het uitvallen van een instantie. Pgpool-II verbreekt al de verbindingen als een enkele instantie offline gaat waarna onmiddellijk daarna de data al opnieuw kan gelezen worden.

De eerste twee systemen zullen een server die opnieuw online komt, automatisch hernemen in hun systeem, bij Pgpool-II moet dit proces handmatig opgestart worden.

# **Hoofdstuk 1**

## **Inleiding**

De hedendaagse meest gebruikte database management systemen (DBMS's) zijn de relationele DBMS's of NoSQL systemen [9]. In deze thesis beschrijft een testmethode om beide systemen te kunnen vergelijken op basis van consistentie en beschikbaarheid, welke daarna toegepast wordt als voorbeeld op MongoDB, HBase en Pgpool-II (een uitbreiding van PostgreSQL).

Het RDBM is er gekomen onder invloed van het artikel van E. Codd over het relationele model in 1969 [7]. Het sleutelconcept van het relationele model is dat de data georganiseerd is in tabellen, gekoppeld door sletuels (constraints). Dit concept leidt tot een vermindering van de redundante data. Voorbeelden van populaire relationele DBMS's (RDBMS's) zijn Oracle, MySQL en PostgreSQL.

De NoSQL databases zijn een nieuwe generatie van systemen, de NoSQL beweging is gestart in 2000 en staat voor '*Not only SQL*'. Deze systemen zijn er gekomen als op de globalisering van de computer systemen. Met een geografische spreiding van de verschillende datacentra konden de RDBMS niet om?. Dit leidde tot de nood voor meer flexibele databases, een lagere complexiteit, hogere doorvoer van data, horizontale schaalbaarheid en het draaien op commodity hardware. NoSQL DBMS proberen hieraan te voldoen met voorbeelden als Google BigTable, Amazon Dynamo, HBase, MongoDB, ... [37]

In de volgende sectie zullen beide systemen in meer detail aanbod komen, waarna de huidige staat voor het kwantitatief vergelijken van de systemen aan bod. Tenslotte zullen de doelstellingen en de bijdragen van de thesis toegelicht worden.

### **1.1 Relationele en NoSQL DBMS's**

Op dit moment zijn de meest gebruikte DBMS's de relationele en NoSQL systemen, maar wat dit net inhoudt en wat de verschillen tussen beiden zijn, zal in deze sectie in meer detail aanbod komen. Eerst zullen het relationele DBMS besproken worden,

## 1. INLEIDING

---

gevolgd door NoSQL een een korte bespreking van de grootste verschillen.

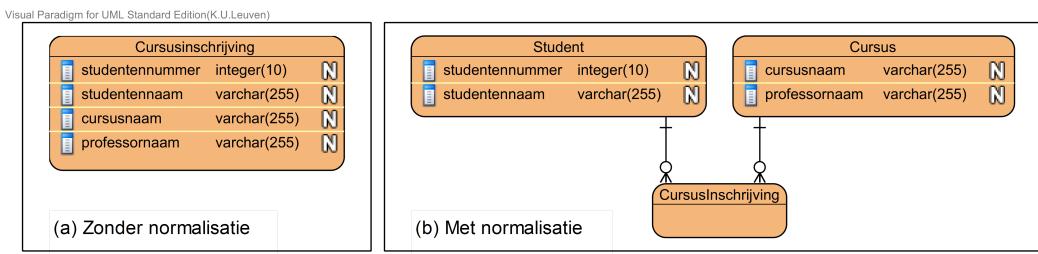
### 1.1.1 Relationale database

Een RDBMS is een DBMS gebaseerd op relationele model voor het structuren van de database.

Het relationele model is vertrekt van theoretische wiskundige principes als set-theorie en eerste-orde predicaten logica. Het model organiseert de data in tabellen en relaties tussen de tabellen. De tabel heeft kolommen die verschillende velden voorstellen waarbij elke rij een collectie van gerelateerde datawaardes is. De relaties tussen de verschillende tabellen toont hoe deze bij elkaar horen. Een belangrijke eigenschap is dat de tabellen en relaties genormaliseerd worden, hiermee wordt redundante informatie verwijderd. Dit zorgt voor een hogere data integriteit en een vermindering in data anomalieën die kunnen optreden bij een update.[12]

De normalisatie kan geïllustreerd worden met het korte voorbeeld van figuur 1.1: de professor voor een vak zal bij elke student hetzelfde zijn, het veranderen van een professor voor een vak zou in het eerste geval een update van alle ingeschreven studenten inhouden, in het tweede geval is dit maar de aanpassing van een enkel record, hetzelfde geldt voor de student.

Interactie met de RDBMS gebeurt op basis van SQL (Structured Query Language), een taal gebaseerd op de relationele logica. SQL geeft uitgebreide query mogelijkheden aan de gebruiker van de software.



Figuur 1.1: Relationeel datamodel (a) zonder en (b) met normalisatie

Een belangrijk concept in een relationele database is ACID, welk voor betrouwbare en robuuste transacties zorgt

**Atomair (Atomicity)** Een database transactie moet oftewel volledig uitgevoerd worden oftewel heeft geen enkele bewerking plaatsgevonden.

**Consistent (Consistency)** Een transactie behoudt consistentie als de volledige uitvoering van de transactie de database van één consistente staat naar een andere brengt. Een consistente staat is een staat die ervoor zorgt dat waarden van een instantie consistent zijn met de andere waarden in dezelfde staat. Een voorbeeld is

het overschrijven van €50 van persoon A naar B, op het einde moet de totale som nog steeds gelijk zijn, A €50 minder en B €50 meer. Een inconsistente staat zou zijn dat enkel A €50 minder heeft, maar B nog steeds evenveel.

**Geïsoleerd (Isolation)** Een transactie moet uitgevoerd worden alsof ze volledig voor of na andere transacties heeft plaatsgevonden.

**Duurzaam (Durability)** Een voltooide transactie kan later niet ongedaan gemaakt worden.

Deze verschillende concepten bieden de garanties welke de gebruiker kan gebruiken voor zijn systeem. Daartegen over staat wel dat dit de complexiteit van de RDBMS groeit, ook indien dit voor bepaalde toepassingen misschien niet nodig is.

### 1.1.2 NoSQL database[37]

NoSQL DBMS zijn ontstaan door een groei en globalisering van de computersystemen en de bijhorende databases. Een RDBMS is gebouwd met een 'one size fits all'-gedachte, maar deze systemen hiermee complexiteit die voor bepaalde toepassingen niet nodig is. NoSQL systemen bestaan in verschillende variëteiten, elk met hun eigen eigenschappen en toepassingsgebied om zo de complexiteit te verminderen. Tussen deze verschillen is er een rode draad te vinden vergeleken met een RDBMS:

- **Lagere complexiteit:** NoSQL systemen bieden minder opties en garanties dan de RDBMS, bepaalde applicaties hebben enkel nood aan een deel van de garanties. Bijvoorbeeld in een sociale netwerk moet een post niet onmiddellijk beschikbaar zijn voor al de vrienden van een persoon, maar dit mag even duren.
- **Hogere doorvoer:** Talrijke NoSQL systemen bieden een hogere doorvoer van data aan wat in veel gevallen een gevolg is van de lagere complexiteit of door de hulp van andere bewerkingen zoals MapReduce [11].
- **Horizontale schaalbaarheid en werkend op commodity hardware:** Waar grote RDBMS's werken met dure high-end systemen, was het bedoeling van NoSQL databases ondersteuning te bieden aan een veelvoud van geclusterde eenvoudige machines (commodity hardware).  
Horizontale schaalbaarheid staat voor het toevoegen extra machines aan een systeem voor extra resources, in tegenstelling tot verticale schaalbaarheid waar een krachtiger machine wordt gebruikt voor de opschaling. De horizontale opschaling wordt tot uitvoering gebracht door de data van een enkele database of tabel te verspreiden over verschillende machines die elk maar voor een deel van de data verantwoordelijk zijn en moeten opslaan.  
NoSQL systemen combineren deze twee elementen en bieden hierdoor een schaalbaar systeem aan met basis componenten.

## 1. INLEIDING

---

- **Datamodel dichter bij objecten:** De meeste NoSQL systemen zijn zodanig ontworpen dat deze de vertaling van objecten naar opslag eenvoudiger maken t.o.v. RDBMS's. RDBMS zijn ontworpen voor het ontstaan van object georiënteerde programmeertalen en heeft de nood aan de vertaling van een object naar de databasestructuur. Bij het ontwerp van NoSQL werd er hiermee onmiddellijk rekening gehouden.

Deze verschillende argumenten leiden vervolgens tot BASE, een tegenreactie op ACID.

- Basis beschikbaarheid (**Basically Availability**): het DBMS biedt lees- en schrijfacties aan bij het falen van één of meerdere falende instanties. De ondersteuning is afhankelijk van systeem tot systeem samen met de configuratie
- Soft State: De data moet op een bepaald moment niet volledig consistent zijn.
- Eventuele consistentie (**Eventual Consistency**): De database zal na enige tijd in een consistente status uitkomen, het is mogelijk dat oudere data tijdelijk leesbaar is. Eventuele consistentie kan op zijn beurt opnieuw onderverdeeld worden in 4 categorieën [22, slide 16]:
  - *Read your own writes* consistentie: Ongeachte van de server waarop een gebruiker leest, zal hij zijn schrijfactie onmiddellijk correct lezen.
  - *Session* consistentie: De gebruiker zal zijn schrijfactie onmiddellijk kunnen lezen binnen dezelfde sessie, een sessie is hierdoor meestal gelimiteerd tot een enkele database server.
  - *Casual* consistentie: Als een gebruiker versie X leest en vervolgens versie Y schrijft, zal elke gebruiker die versie Y leest ook versie X lezen.
  - *Monotonic Read* consistentie: Dit levert monotone tijdsgaranties dat een gebruiker enkel recentere data versies in de toekomst zal lezen.

De BASE eigenschappen kunnen gekoppeld worden aan de CAP theorie van Erik Brewer[4]. CAP zegt dat een gedistribueerd systeem maar twee van de 3 CAP elementen kan ondersteunen: consistentie, beschikbaarheid en partitie tolerantie. De beschikbaarheid betekent dat bij het falen van een instantie er nog steeds schrijfbewerkingen mogelijk zijn. Bij partitie tolerantie kan het systeem overweg met het opgesplitst zijn van instantie door een niet werkende netwerk verbinding. De definitie van consistentie is hier anders als bij ACID: bij CAP is er sprake van consistentie als het DBMS zich gedraagt alsof er maar een enkel kopie van de data is.

### Classificatie van NoSQL systemen

Er zijn vele NoSQL systemen ontworpen gedurende de laatste jaren, elk met hun eigen variëteit, functionaliteit en populariteit. Er bestaan verschillende manieren om

de systemen te classificeren, maar één van de meest gebruikte doet dit op basis de data modellering. Een korte vergelijking op basis van deze bevindt zich in tabel 1.1.

Soort	Performantie	Schaalbaarheid	Flexibiliteit	Complexiteit	Functionaliteit
Column	hoog	hoog	gematigd	laag	minimaal
Document	hoog	variabel(hoog)	hoog	laag	variabel (laag)
Graph	variabel	variabel	hoog	hoog	graph theory
Key-Value	hoog	hoog	hoog	geen	variabel (geen)

Tabel 1.1: Classificatie en categorisatie van NoSQL DBMS's door Scofield en Popescu. [36] [32]

**Column Model** In een column-gebaseerd systeem wordt de data opgeslagen per kolom in plaats van de traditionele manier, per rij. Deze aanpak werd in eerste instantie gedaan voor analyse van business intelligentie. Het systeem is geïnspireerd door de paper van Google's Bigtable [6]. [37]

**Graph Model** In een grafen model, wordt de data voorgesteld en opslagen volgens de grafen theorie: knopen, lijnen en eigenschappen op de knopen en lijnen. [2].

**Key-Value Model** Key-Value systemen hebben een heel eenvoudig data model, data kan opgeslagen, opgevraagd en verwijderd worden op basis van een key. De informatie die in de database zit, is de waarde voor die key.

Met dit eenvoudig model en functionaliteit die weinig complexiteit introduceren, kan er gestreefd worden naar een hoge performantie, schaalbaarheid en flexibiliteit. [37]

**Document Model** Document systemen zijn volgens vele de volgende stap in key-value systemen, waar deze complexere structuren toe laten, dit door middel van meerdere key/value paren per element. [37]

Een document moet geen vaste structuur hebben maar elk document op zich kan verschillende velden hebben, dit kan bijvoorbeeld toegepast worden bij boeken. Waar een bepaald boek een recept is, kan een ander een deel zijn van een trilogie. Bij het eerste kan de kooktijd opgeslagen worden en bij de tweede een referentie naar de andere boeken. [37]

### 1.1.3 Bespreking van verschillende DBMS's

Databases uit 4 categorieën komen verder aanbod, er is gekozen om de Graph NoSQL DBMS's niet te bespreken. Graph NoSQL DBMS's zijn bedoeld voor de opslag van data van grafen. Deze is significant verschillend van de andere categorieën en hierdoor niet opgenomen.

- Column NoSQL DBMS's: Cassandra, HBase
- Document NoSQL DBMS's: Apache CouchDB, MongoDB

## 1. INLEIDING

---

- Key-Value NoSQL DBMS's: LightCloud (Tokyo), MemCache, Redis, Riak, Project Voldemort
- Relationale DBMS's: MySQL, Pgpool-II (PostgreSQL)

Deze keuze van deze systemen is gebaseerd op de paper van Christophe Strauch [37]. Een korte bespreking van de verschillende systemen kan gevonden worden in bijlage A.

## 1.2 Vergelijking van DBMS's naar performantie en CAP

Bij de ontwikkeling van verschillende systemen is er een keuze naar welk DBMS er gekozen wordt. De systemen verschillen en hebben elk hun eigen toepassingsgebied. Zoals besproken hierboven kan een opsplitsing naar het datamodel gemaakt worden, of in meer detail naar de ondersteunde database bewerkingen.

Maar de systemen kunnen ook verschillende performantie of een keuze in het CAP theorema gekozen hebben. In dit gedeelte zal er gekeken worden welke methodes er al beschikbaar zijn voor het kwantitatief vergelijken van de performantie, consistentie en beschikbaarheid en mogelijke resultaten.

### 1.2.1 Performantie benchmarking

Indien men verschillende DBMS's wilt vergelijken bestaan er al enkele tools en studies om de performantie te kunnen vergelijken. Een blogpost van A. Popescu [31] geeft een overzicht van verschillende benchmarking tools.

Als eerste hebben vele DBMS's **interne benchmarking tools**, waarmee de database op verschillende configuraties kan getest en vergeleken worden. Deze resultaten zijn nuttig na de keuze van het DBMS. Het systeem kan getest worden bij het variëren van de parameters en het uitzoeken wat de bottleneck is in een bepaald systeem. Een voorbeeld hiervan is mongoperf<sup>1</sup> voor MongoDB.

Andere studies focussen op het testen van verschillende systemen en daarbij kunnen verschillende doelstellingen zijn: het ontwikkelen van een breed toepasbare tool, het testen van een grote verscheidenheid van DBMS's of het testen van een specifieke categorie van systemen. Elke van deze benchmarking brengt nieuwe kennis van de systemen maar heeft ook zijn beperkingen. Het totaal pakket van al de testen kan een gebruiker de informatie geven om een beter gefundeerde keuze te maken.

Een eerste categorie van deze externe tools is het **ontwikkelen van een tool** voor verschillende systemen. Dit heeft als grote voordeel dat andere gebruikers nadien

---

<sup>1</sup><http://docs.mongodb.org/manual/reference/program/mongoperf/>

de testen opnieuw kunnen uitvoeren met de systemen in hun configuratie. Het is namelijk niet gegarandeerd dat het resultaat van een jaar geleden gelijkaardig is met de nieuwste versie. Het grootste nadeel is de testen die kunnen uitgevoerd worden, er is een grote variëteit aan systemen elk met hun eigen datastructuur en query mogelijkheden. De tool moet dus een gemeenschappelijke subset zoeken en enkel dit soort queries kunnen getest worden. Een voorbeeld van een dergelijke tool is YCSB[8]. Deze tool kan elk DBMS's testen zolang een basisset van 5 queries ondersteund wordt: het invoegen, updaten, verwijderen en opvragen van een enkel record met daarnaast ook de mogelijkheid tot scan queries, met behulp van 1 query een verzameling van records tegelijk op te vragen.

Sommige systemen ondersteunen bepaalde queries niet rechtstreeks maar bevatten wel de functionaliteit om deze met behulp van meerdere achtereenvolgende bewerkingen te implementeren. Bijvoorbeeld een update kan geïmplementeerd worden door het opvragen, verwijderen en vervolgen invoegen van het aangepaste record.

Een volgende categorie zijn de **resultaten van gerelateerde DBMS's**, dit zijn voornamelijk systemen met hetzelfde datamodel. Het grote voordeel hieraan is dat deze systemen in de meeste gevallen een vrij gelijkaardige set aan query mogelijkheden bevatten waardoor er meer diepgang is dan tussen meer verschillende systemen. Een voorbeeld van zulk onderzoek is gedaan door P. Pirzadeh et al[30] voor de key-value systemen, meer specifiek is er gefocust op het uitvoeren van range queries tussen Cassandra, HBase en Voldemort.

In deze categorie vallen ook de resultaten die meestal getoond worden op de website van de DBMS's, een vergelijkende benchmark met andere soortgelijke systemen. Hoewel de resultaten niet altijd volledig objectief zijn, kan de gevolgde test methode wel interessant zijn. Een voorbeeld van deze studie is de Key-Value benchmarking van VoltDB[17] waar Cassandra en VoltDB vergeleken worden, een belangrijke kanttekening is dat de auteur zelf al aanhaalt dat de systemen vrij verschillend zijn.

Als laatste categorie, zijn er de **resultaten van verschillende DBMS's** waar verschillende soorten systemen met elkaar getest worden. De belangrijkste voordeel is dat er resultaten zijn die verschillende soorten met elkaar vergelijken en waardoor niet alleen verschillen in het datamodel kunnen vergeleken worden in toekomstige studies maar ook performantie verschillen. Het nadeel is dat er een gemeenschappelijke subset gevonden moet worden, hierdoor kunnen bepaalde databases hun kracht net niet laten zien. Enkele van deze onderzoeken zijn [39] en [34]. Deze laatste maakt gebruik van de YCSB tool die hierboven besproken was.

### 1.2.2 Consistentietesten

Bij een gedistribueerd systeem kunnen er verschillende keuzes gemaakt worden naar synchrone of asynchrone replicatie en welk soort consistentie er aangeboden wordt. In de documentatie van DBMS's worden er beloftes gemaakt, maar hoe is de consistentie in de realiteit?

## 1. INLEIDING

---

Een recent artikel [15] (maart 2014), stelt dat er momenteel nauwelijks gekwantificeerde methodes bestaan om de eventuele consistente te meten. In hun artikel stellen zij twee mogelijke methoden voor: de actieve of passieve analyse.

De **actieve** analyse bestaat uit het wegschrijven van data waarna men hoe lang het duurt vooraleer alle servers de nieuwe waarde hebben. Bij de **passieve** analyse kijkt men langs de gebruikerskant. Leest de gebruiker altijd de laatste waarde (=strikte consistentie)? Is het mogelijk dat een nieuwe waarde al wordt gelezen voor de schrijfactie voltooid is?

Beide analyses hebben hun eigenschappen, de actieve analyse is gericht op het database systeem en zijn server. Bij de passieve analyse is georiënteerd naar de gebruiker toe, hoe moet deze zijn toepassingen aanpassen, wat zijn de garanties die geleverd worden aan de gebruiker?

Voornamelijk naar actieve analyse is er al kwantitatief onderzoek verricht. Onder andere Duitse onderzoekers hebben op het Amazon S3 platform getest hoe lang het duurt vooraleer data geschreven in MiniStorage beschikbaar is voor alle gebruikers op al de verschillende servers. [1].

Daarnaast zijn er ook 2 interessante resultaten gevonden: allereerst heeft het Amazon S3 systeem geen monotone lees consistentie, daarnaast bleek het inconsistentie interval voor een bepaald record periodiek verloop te hebben dat niet door de onderzoekers verklaard konden worden.

De YCSB software van hierboven is door onderzoekers in de VS uitgebreid naar YCSB++[28] waardoor deze meer ondersteuning heeft voor het meten van systeembelasting maar ook voor de consistentie-eigenschappen. Enkele geteste systemen zijn in principe strikt consistent, zoals HBase, maar deze worden eventueel consistent door het gebruiken van buffers bij de gebruiker. Vervolgens testen zij hoe lang het duurt voor de data ook gelezen kan worden. De vertraging is sterk afhankelijk is van het aantal acties van de schrijvende gebruiker: indien er meer geschreven wordt, zal de buffer sneller verzonden worden naar de server en dus sneller beschikbaar zijn voor andere gebruikers.

Hoewel zij stellen dat er ook testen zijn gedaan naar eventuele consistentie voor Cassandra en MongoDB, zijn de resultaten niet beschikbaar in het artikel of op de website.

Andere onderzoeker[41] doen analyse op Amazon SimpleDB. In de situatie wordt getest of de database read-your-own-writes en monotone consistentie ondersteund. Aan het eerste is niet voldaan met *eventual consistency read*. Met minder dan 500 ms tussen het einde van de schrijfactie en het begin van de leesactie, wordt er slechts 33% van de nieuwe data gelezen, zodra er meer als 500 ms gewacht wordt, gaat dit naar 99%. Ook is er geen monotone consistentie omdat er van verschillende servers kan gelezen worden bij opvolgende leesacties, sommige zullen de data al wel hebben, anderen niet.

Bij Netflix heeft men aan passieve analyse gedaan op hun Cassandra systeem [20]

waar zij in hun testen geen consistentieproblemen vonden naar de gebruiker toe. Er is geen vermelding hoeveel vertraging er zit tussen beide transacties. Volgens hun gaat het meer om de perceptie dat data verkeerd kan gelezen worden en de angst van het middle management.

#### 1.2.3 Beschikbaarheidstesten

Een derde verschilpunt is hoe de systemen omgaan met het falen van een enkele server en dit onder verschillende opties: Het is mogelijk dat deze tijdelijk uitgeschakeld wordt wegens onderhoud. Het kan gaan om een onverwachte crash van de software op een server of een crash van een volledige server, tenslotte kunnen er ook nog netwerkproblemen optreden waardoor deze (tijdelijk) niet beschikbaar is.

Nu hoe gaan deze systemen om het falen en terug online brengen van de systemen? Zijn er geen acties mogelijk op de server, worden de connecties tijdelijk verbroken, is er een verhoogde of verlaagde vertraging op de transacties? Detecteert het systeem automatisch wanneer de oorspronkelijke server terug online komt of moet gemeld worden om de server terug te gebruiken? In een NoSQL DBMS waar gewerkt wordt commodity hardware, zal het falen regelmatig gebeuren en verschillende systemen reageren anders op deze acties.

Uit de literatuurstudie zijn er geen vergelijkende studies voor database systemen gevonden. Er is voor de meeste DBMS's informatie te vinden op de website hoe zij in een gedistribueerde omgeving werken zoals het al dan niet gebruik van sessies en de duur van een sessie. Maar voor het effect in de praktijk, is het handmatig testen.

## 1.3 Doelstelling en bijdrage

In de literatuurstudie komt naar voor dat er een gebrek is aan kwantitatieve methodes om DBMS's te vergelijken naar consistentie en beschikbaarheid.

Loop hier nog is over

Het eerste doel van deze thesis is om benchmark te ontwikkelen die het gedrag in de praktijk kan testen. Deze testen zullen het uitvalen van een service, server en netwerk simuleren voor de beschikbaarheidstesten. Bij de consistentietesten wordt er een tool aangeleverd die verschillende types van eventuele consistentie kan testen.

Daarnaast zal aangetoond worden dat de testmethode werkt door het toepassen op drie voorbeeldsystemen: HBase, MongoDB en Pgpool-II(uitbreiding van PostgreSQL). Door het kiezen van verschillende systemen met een verschillend datamodel, kan de testmethode een kwantitatieve vergelijking ten opzichte van elkaar mogelijk maken.

Tenslotte is het doel om de testen eenvoudig te kunnen uitvoeren voor andere gebruikers, op deze manier kan de data gecontroleerd worden. Daarnaast wordt er ook de mogelijkheid aangeboden om nieuwere versies en andere infrastructuren te kunnen testen.

## **1. INLEIDING**

---

Met deze drie doelstellingen wordt er een testmethode, een werkende benchmarking tool en resultaten aan een gebied waar er nog nauwelijks kwantitatieve methodes zijn.

### **1.4 Conclusie**

Deze thesis zal handelen over het ontwikkelen en uitvoeren van een benchmark systeem voor consistentie en beschikbaarheid naar relationele en NoSQL databases. Momenteel zijn er weinig kwantitatieve methodes en resultaten naar beschikbaarheid en consistentie.

In het vervolg van deze tekst, zal in hoofdstuk 2 een algemene testmethode voor consistentie en beschikbaarheid voorgesteld worden. In hoofdstuk 3 wordt besproken hoe de voorgestelde methode in de praktijk geïmplementeerd wordt. Daarna worden de observaties voorgesteld in hoofdstuk 4 met een analyse en verklaring van de observaties in hoofdstuk 5. Een conclusie volgt tenslotte in hoofdstuk 6.

## Hoofdstuk 2

# Voorgestelde testmethodiek

Dit hoofdstuk behandelt de wijze waarop de testen naar consistentie en beschikbaarheid worden uitgevoerd. De methodiek is opgedeeld in 4 grote stappen: het opstellen, kalibreren, testen van de systemen en tenslotte het verzamelen en analyseren van de resultaten. Een overzicht van de procedure kan gevonden worden in figuur 2.1.

**Opstellen van de testomgeving** Deze eerste stap is voor het selecteren, installeren en configureren van een DBMS en de testsoftware. Een variatie in hardware van de systemen, versienummer van de software of een verschillende netwerkinfrastructuur kan de uiteindelijke testresultaten beïnvloeden.

**Calibratie van de testomgeving** In de uiteindelijke testen wordt het gedrag onder matige belasting getest. Afhankelijk van de gekozen systemen, netwerkinfrastructuur zal dit voor elke DBMS een verschillende belasting geven. Deze stap bepaalt welke queries er uitgevoerd worden, hoeveel gebruikers er zijn in het systeem en hoeveel bewerkingen er uitgevoerd worden per second.

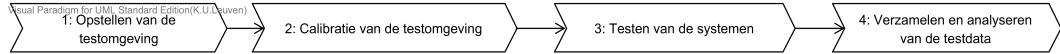
**Testen van de systemen** In deze stap worden de testen op de verschillende systemen uitgevoerd. Deze methodiek maakt het mogelijk om te testen hoe de vertraging op een bewerken zich gedraagt voor, tijdens en na het falen en herstellen van een systeem. Voor de consistentie wordt het mogelijk om zowel aan passieve als actieve analyse te doen.

**Verzamelen en analyseren van de testdata** In de laatste stap wordt de data van de vorige stappen verzameld en de resultaten worden visueel voorgesteld. Met behulp van de uitgebreide testdata, is het ook mogelijk om bepaalde conclusies te maken over een de beschikbaarheids- en consistentiegaranties van de verschillende.

In de volgende secties komen de verschillende stappen in meer detail aan bod.

## 2. VOORGESTELDE TESTMETHODIEK

---



Figuur 2.1: Overzicht testproces

### 2.1 Stap 1: Opstellen van de testomgeving

**Keuze van DBMS's** In eerste instantie moet er gekozen worden welke DBMS's er getest worden. Er zijn verschillende mogelijke keuzes, er kan een enkele systeem getest worden in één of meerdere configuraties of verschillende systemen.

**Bepalen van de infrastructuur** Na van de systemen kan de configuratie van de testomgeving gebeuren. Dit gebeurt eerst door het aantal instanties per systeem te beslissen en de hardware te kiezen.

**Installatie en configuratie** Het lokaal installeren en configureren een software-pakket, is in Unix veelvuldig geautomatiseerd met behulp van tools zoals *apt-get* en *yum*. Voor een systeem in een gedistribueerde omgeving, is de situatie ingewikkelder. In een gedistribueerd systeem, dienen de verschillende servers van elkaar op de hoogte gebracht.

In een gedistribueerde configuratie stap worden de verschillende systemen van elkaar bestaan op de hoogte gebracht en worden de relaties opgezet. Hiervoor bestaan er twee verschillende methodes maar ook een combinatie is mogelijk.

**Configuratie bestanden** Met deze methode heeft elke lokaal systeem een configuratiebestand met hierin een link naar één of meerdere andere instanties. Nadat de administrator deze configuratie heeft aangemaakt, kan het DBMS gestart worden. Deze informatie kan een ip adres of de hostname zijn waarbij soms 1 instantie voldoende is, bij andere moeten ze allemaal opgegeven worden.

**Centrale configuratie** Bij een centrale configuratie, worden de systemen lokaal opgestart zonder informatie van de andere instanties. Vervolgens wordt via een console, webinterface, ... connectie gemaakt met een node. Deze krijgt configuratie informatie hoe deze zich moet gedragen en volgt deze informatie op. In deze systemen is de configuratie tijdens installatie gelijk en wordt de configuratie verspreid wanneer de systemen al draaien.

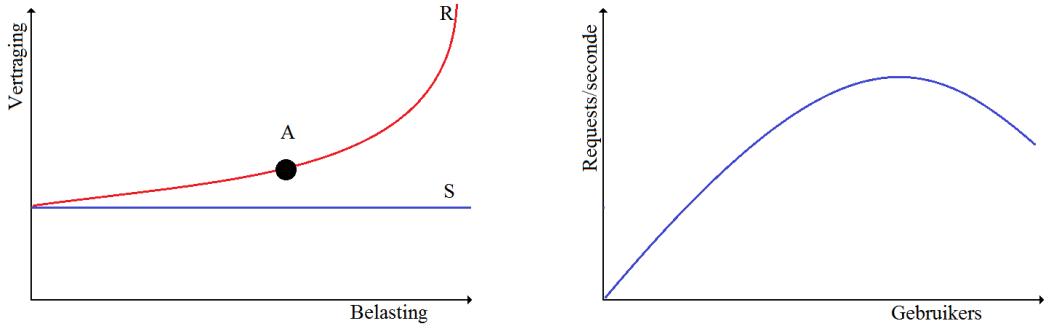
Na het uitvoeren van deze stap, zou het DBMS moeten werken zoals vereist.

### 2.2 Stap 2: Calibratie van de testomgeving

Afhankelijk van de onderliggende infrastructuur en het soort DBMS, kan het systeem een verschillend gedrag hebben onder dezelfde configuratie. Voor de eigenlijk testen

## 2.2. Stap 2: Calibratie van de testomgeving

is het de bedoeling om een middelmatige belasting te hebben. De queueing theorie geeft de eigenschap dat de totale vertraging ( $R$ ) gelijk is aan de som van verwerkings- ( $S$ ) en wachttijd ( $W$ )[23]. Dit verband is visueel voorgesteld ten opzichte van de belasting in figuur 2.2(a).



Figuur 2.2: Verbanden voor de calibratie

Deze belasting kan afhankelijk zijn verschillende elementen, er worden 5 verschillende mogelijke parameter groepen besproken:

**Hoeveelheid data per gegevensrecord** Elke record in de database kan bestaan uit verschillende kolommen en per kolom een waarde. Het is belangrijk om te definiëren hoe groot een gemiddeld record is, het type data in een record en het aantal kolommen. Een klein record zorgt voor minder netwerkverkeer en een ander schrijfgebruik dan bij een groot record.

**Type van queries** De opgeslagen data kan opgevraagd worden op verschillende wijze. Data kan ingevoegd, aangepast, opgevraagd of verwijderd worden. Daarnaast kan dit gebeuren voor 1 of meerdere records tegelijk. Afhankelijk van de relatieve verhouding van deze soorten, kan een ander resultaat bekomen worden: sommige DBMS's zijn meer geschikt voor een dominantie in leesacties andere voor schrijfacties. Enkele systemen lezen data goed in grote hoeveelheid, andere lezen zeer goed in kleine hoeveelheden.

**Query specificatie** Bij het opvragen of verwijderen van een record, kan er een verschil zijn naar processing tijd afhankelijk van hoe lang geleden de record geschreven of gelezen is en of naburige data onlangs gelezen is. Vandaar dat ook het datadistributie gekozen moet worden. Voorbeelden van verschillende technieken zijn: voornamelijk de laatste data lezen, een uniforme kans voor alle data of bepaalde records regelmatig lezen.

**Aantal connecties of gebruikers** Een gedistribueerde omgeving heeft meestal meerdere gebruikers die tegelijk actief zijn. Maar sommige systemen hebben een voorkeur naar weinig connecties met grote hoeveelheden data, andere kunnen meer gebruikers tegelijk behandelen. Het totaal aantal queries kan berekend worden als:  $\#Queries = \#Gebruikers * \#QueriesPerGebruiker$ . In deze stap wordt er verondersteld dat de gebruiker het maximaal aantal queries doet, dus  $1/Vertraging$ . Rekening houdend met de exponentiële groei van de wachtrij vertraging (figuur 2.2(a)), betekent dit dat er een maximum aantal queries per seconde bereikt wordt bij een bepaald aantal gebruikers. In deze stap wordt er gezocht naar dit aantal gebruikers. De grafiek zal er meestal uitzien zoals in figuur 2.2(b).

**Aantal queries per seconde** In de vorige stap is er de optimale configuratie bepaald om het systeem maximaal te beladen. In het begin is er gesteld dat er gezocht wordt naar een gemiddelde belasting voor dit aantal gebruikers. Er wordt gekozen om matige belasting, in figuur 2.2(a) zou dit punt A zijn.

Met de parameters afkomstig uit de calibratie, kunnen de testen opgestart en uitgevoerd worden.

### 2.3 Stap 3: Testen van de systemen

In deze thesis zullen er 2 verschillende soort testen uitgevoerd worden, de beschikbaarheid en consistentietesten, welke dezelfde algemene stappen volgen, elk met hun eigen specifieke parameters. Er zijn de 6 deelstappen:

**Opstellen van de database** In stap 2 was er gekozen voor een bepaalde datastructuur. Deze structuur wordt zo goed mogelijk ingesteld in de DBMS zodat deze optimale allocatie kan doen.

**Inladen van de data** Een bepaalde hoeveel data wordt vooraf ingeladen. Dit wordt gedaan om een basis hoeveelheid data te hebben die nodig is voor de initialisatie van de database. Zo wordt er met deze dataset sharding toepast, het opsplitsen van de data over verschillende servers. In bepaalde DBMS's wordt data automatisch opgesplitst bij het groeien van de dataset, om deze reden wordt er data ingeladen zodat deze automatische sharding gebeurt. Dit inladen van de data gebeurt op maximale snelheid.

**Pauze** Na het inladen van de data wordt enige tijd gewacht. Zoals aangetoond in YCSB++[28, Figuur 9], is er hogere vertraging in de DBMS's onmiddellijk na het inlezen. Dit kan onder andere te wijten zijn doordat data nog weggeschreven moet worden naar schijf of in bepaalde systemen zou het kunnen dat de sharding gebeurt op momenten met weinig belasting. Met het toevoegen van de wachtperiode wordt de piek in de vertraging vermeden.

## 2.3. Stap 3: Testen van de systemen

---

**Opstarten van de test (opstart kost)** De test wordt opgestart. In veel gevallen is er in het begin een opwarmfase nodig omdat de vertraging net hoger of lager is als na enige tijd. Deze hogere tijd is onder andere te verklaren door de connecties opgezet moet worden en caches voor gelezen data worden gevuld. Soms is deze lager omdat de schijf nog niet belast is of de er nog veel schrijfbuffers leeg zijn. Om dit gedrag te vermijden, wordt de data verzameld de eerste seconden niet gebruikt voor de analyse.

**Uitvoeren van de test** De eigenlijke test wordt uitgevoerd, de data wordt verzameld en opgeslagen. De details van de beide testen volgen achteraf.

**Terugbrengen naar beginstatus** Na het uitvoeren van de test, wordt het DBMS terug naar de beginstatus gebracht. Onder andere de database en de data wordt volledig verwijderd. Belangrijk in dit geval is het controleren of de data volledig verwijderd is, in bepaalde gevallen wordt er nog ergens een veiligheidskopie bijgehouden dat een volgende test kan beïnvloeden.

De twee verschillende testmethodes zullen nu in meer detail behandeld worden.

### 2.3.1 Beschikbaarheidstest

Bij de beschikbaarheidstest wordt er gekeken hoe het systeem reageert op tijdelijke, (on)verwachte onbeschikbaarheid van een deel van het systeem. In deze testen worden er 3 mogelijke manieren getest die de systemen onbeschikbaar maakt, terwijl er de belasting uit de calibratie wordt toegepast.

**Zachte stop** De DMBS service wordt gevraagd om te stoppen. Op deze manier krijgt de service eerst een signaal dat deze moet stoppen en kan deze de andere waarschuwen. Achteraf wordt dezelfde service terug opgestart. Dit simuleert het gepland uitschakelen van een systeem.

**Harde stop** De DMBS service wordt onmiddellijk gestopt door het process te beëindigen. De service heeft geen tijd om de andere te waarschuwen. Achteraf wordt dezelfde service terug opgestart. Dit simuleert een crash van de service die de systeembeheerders later opmerken en de service opnieuw opgestart wordt.

**Netwerk onderbreken** Al het in- en uitgaand netwerkverkeer wordt gestopt zonder enige waarschuwing. De service heeft geen tijd om de andere te waarschuwen én de zender krijgt geen onbereikbaar antwoord. Achteraf wordt het netwerk verkeer terug toegelaten. Dit simuleert een onderbroken internetverbinding of een gecrashte server.

Eenzelfde systeem kan sterk verschillend reageren op de verschillende situaties: waar de eerste situatie nog eenvoudig is te behandelen doordat het systeem de andere

## **2. VOORGESTELDE TESTMETHODIEK**

---

op de hoogte kan brengen. In de tweede situatie wordt het moeilijker omdat het de andere systemen niet op de hoogte kunnen gebracht worden. Maar de systemen krijgen bij het contacteren het antwoord dat de service onbeschikbaar is. De derde situatie is het moeilijkste te behandelen omdat men niet weet of de berichten naar de server niet aankomen of de antwoorden verloren gaan.

In dit geval kan er onderzoek gedaan worden naar het verschil in vertraging en de beschikbaarheid van de laatst geschreven data elementen. In dit onderzoek is er enkel gefocust op de reactie naar de vertraging toe.

### **2.3.2 Consistentietest**

In de consistentietest wordt onderzocht welke consistentie het DBMS ondersteund. Zoals voordien besproken in deel [1.1.2](#), bestaan er verschillende soorten.

In deze testen is er gekozen om caching bij de gebruiker **uit te schakelen**, dit om de reden dat dit gedrag zeer onvoorspelbaar is en afhankelijk van andere acties van de lezer en schrijver. Een andere reden is dat eventueel consistentie alleen een probleem is voor data die onmiddellijk beschikbaar moet zijn, met andere woorden data die men niet mag cachen. Dit heeft als gevolg dat de belasting op de server hoger kan zijn.

**Beschrijving van de test** Deze test bestaat uit 3 soorten gebruikers: er is een gebruiker die data schrijft (=S), een aantal lezer (=L's) en tenslotte zijn er nog andere gebruikers die zorgen voor de basisbelasting. De berekening van deze basisbelasting komt verder aanbod. Het is belangrijk dat er een exacte synchronisatie in tijd is tussen de verschillende gebruikers, dit om de geregistreerde tijdstippen te kunnen vergelijken.

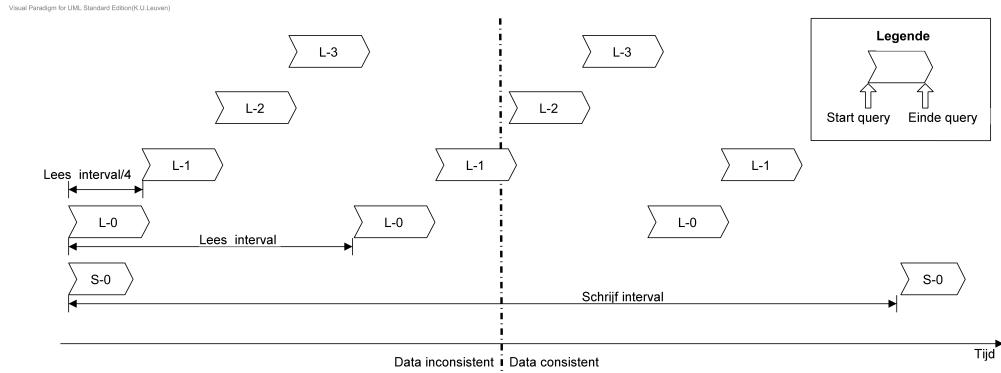
**Taak van de schrijver** De schrijver schrijft, zoals zijn naam voorspelt, voorafbepaalde data weg op vooraf vastgelegde momenten. De data kan een nieuw record of een update van een record zijn. De schrijver registreert op welk exact tijdstip deze taak is gestart, hoe en wanneer deze is beëindigd.

**Taak van de lezer** De taak van een individuele lezer is om op vooraf vastgelegde momenten de data van de schrijven te gaan lezen. Dit wordt periodiek herhaald tot de data correct is gelezen of een bepaalde tijd is verstreken. Er kan ook beslist worden om ook als de data correct gelezen is, te blijven lezen om te zien of het resultaat terug veranderd. De lezer registreert elke keer deze gaat lezen op welk moment deze exact is gaan lezen en wat het resultaat van de actie is.

**Het plannen van de lezers** Het doel van de verschillende lezers is om allereerst verbonden te zijn met verschillende servers zoals ook gebruikers zullen zijn. Daarnaast zijn er meer testpogingen voor het lezen van de data omdat verschillende lezers in

### 2.3. Stap 3: Testen van de systemen

parallel kunnen lezen. Om deze laatste redenen worden de starttijdstippen voor de data te lezen, gelijk gespreid tussen de verschillende lezers. Een voorbeeldperiode met 1 schrijver en 4 lezers kan gevonden worden in figuur 2.3.



Figuur 2.3: Consistentietest: Een enkele periode van de consistentietesten. Er is 1 schrijver, 4 lezers. De lezers stoppen zodra deze de data correct hebben gelezen. De rode lijn geeft aan vanaf wanneer de data consistent is voor alle queries gestart na dit tijdstip.

**Schatten van de basisbelasting** De basisbelasting kan de berekende belasting zijn in stap 2, waardoor het reëel aantal queries hoger ligt. De belasting kan ook verminderd worden met een geschat aantal queries die de schrijvers en lezers zullen uitvoeren. Het aantal queries van de schrijver en lezers per seconde kan berekend worden aan de hand van de hand van volgende formule:  $(S + \#L * \#queriesperschrijfperiode / schrijfinterval)$ . Het aantal leesbewerkingen per schrijf periode zal geschat moeten worden, maar kan bijvoorbeeld op 1 gezet worden. Op deze manier krijgen systemen die geen strikte consistentie afdwingen een hogere belasting om de correcte waarde te lezen. Er zal immers een tweede keer of meer geprobeerd moeten worden.

**Soorten eventuele consistentie** Met deze uitgevoerde testen en data kan aangegetoond worden dat bepaalde systemen bepaalde eventuele consistentie vereisten niet volgen. Het is in mogelijk om met deze testen een tegenvoorbeeld te vinden zie tonen dat een systeem een eigenschap niet garandeert. Maar het vinden van geen tegenvoorbeeld is geen bewijs, maar kan wel een indicatie zijn hoe vaak het kan voorkomen.

**Consistentie** Een systeem is niet consistent indien één van de lezers het nieuwe record of de update niet leest *indien de leesactie gestart is na het voltooiien van de schrijfactie*. Als dit voorkomt, gedraagt het systeem zich niet alsof er maar een enkele data-instantie is.

## 2. VOORGESTELDE TESTMETHODIEK

---

**Read your own writes consistentie** Deze eventuele consistentie kan ontkracht worden indien een schrijver onmiddellijk na het voltooien zijn eigen data opvraagt en niet de nieuwe waarde leest. Voor deze testen dient de server ondersteuning te bieden om meerdere connecties te hebben per gebruiker.

**Session consistentie** Session consistentie is een verzwakking van de vorig eis. Het is nu slechts nodig om de data te lezen van een schrijfactie in eenzelfde sessie, niet voor nieuwe sessies van dezelfde gebruiker. Dit kan ontkracht worden door met dezelfde connectie als de schrijver te lezen en de oude data te lezen.

**Casual consistentie** Deze test kan uitgevoerd worden indien de schrijver verschillende schrijfacties na elkaar te laten uitvoeren. De lezer leest de records in dezelfde of omgekeerde volgorde. Indien deze data van een latere schrijfactie leest maar nog niet van een vroegere, is dit ongeldig. De eis kan strenger gemaakt worden door de schrijver tussendoor niet te laten lezen, dit zou andere resultaten kunnen hebben. Deze consistentie is niet getest.

**Monotonic Read consistentie** In deze test blijft de lezer continue opnieuw proberen om dezelfde data te lezen. Eenmaal deze een nieuwe versie heeft gelezen, zou deze nooit meer oudere data mogen lezen.

Zoals duidelijk hierboven, biedt deze aanpak de mogelijkheid aan om naast een actieve ook een passieve analyse te doen op de data. In deze thesis zal er gefocust worden op de *read your own writes* en *monotonic read* consistentie.

### 2.4 Stap 4: Verzamelen en analyseren van de testdata

Na het uitvoeren van de testen, dient de informatie die verschillende schrijver en lezers hebben vergaard, samen gebracht te worden. Met de verwerking van deze informatie wordt bepaald hoe lang het duurt voor de data overall consistent is of om tegenvoorbeeld te zijn voor een bepaalde consistentie categorie. Bij de beschikbaarheidstesten wordt er onderzocht worden hoe lang bepaalde bewerkingen niet mogelijk zijn en of meer of minder systemen een invloed heeft op de vertraging.

Voor de testen worden verschillende grafieken gegenereerd van de aanwezige data, dit met de voor de hand liggende reden dat een figuur meer duidelijkheid brengt over de data dan duizenden getallen.

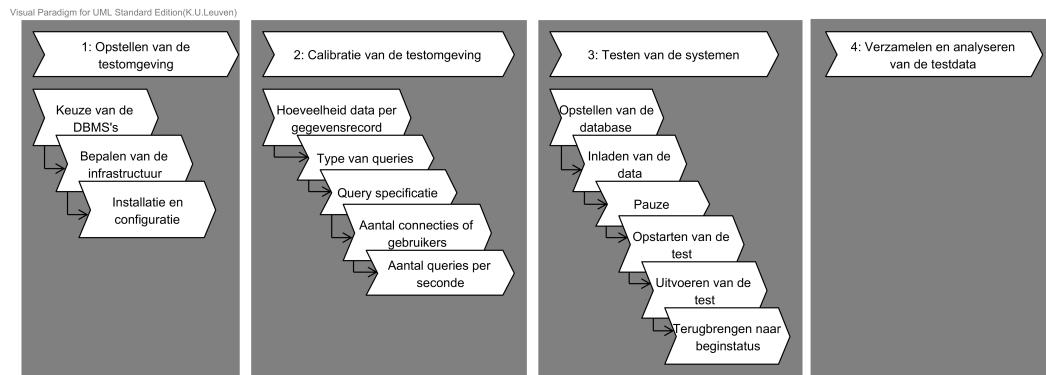
### 2.5 Conclusie

In dit hoofdstuk werden de verschillende stappen van de testmethode besproken. In het totaal bestaat de test uit 4 verschillende stappen. Een overzicht van de test methode kan gevonden worden in figuur 2.4.

## 2.5. Conclusie

---

Deze testmethode biedt de mogelijkheid om op een vergelijkbare manier verschillende systemen te testen naar hun gedrag in consistentie en beschikbaarheid. In het volgende hoofdstuk zal de implementatie besproken worden.



Figuur 2.4: Overzicht testproces



# Hoofdstuk 3

## Implementatie

In het vorige hoofdstuk is uitgelegd wat de methode is voor het testen, maar hoe vertalen deze vereisten naar een werkende test programma? In dit hoofdstuk zal deze vertaling uitgelegd worden die op deze thesis is toegepast.

In het eerste deel wordt uitgelegd wat de selectie criteria waren om HBase, MongoDB en Pgpool-II (PostgreSQL) te verkiezen. Daarna zullen drie systemen in meer detail uitgelegd worden om zo de specifieke architectuur uit de doeken te doen. Daarna wordt de testsoftware besproken met de testconfiguratie.

meer?

### 3.1 Selectie van de DBMS's

Voor de selectie van de systemen is er onderzocht of een systeem een bepaalde eigenschap al dan niet ondersteunt. In het totaal zijn er 5 verschillende eigenschappen waarop de selectie is gebaseerd.

**Vrije software** Om testen tussen verschillende DBMS's te kunnen vergelijken op een gelijkaardige infrastructuur, is het nodig dat deze software kan geïnstalleerd worden op de eigen infrastructuur. In deze thesis is er gefocust op systemen die gratis aangeboden worden.

**Persistentie** Voor het testen van de beschikbaarheid van de data, is het een voordeel dat de data op harde schijf aanwezig is: bij een herstel dient er minder data over het netwerk gestuurd te worden. Om deze reden hebben persistente systemen een voorkeur op deze die de data enkel in geheugen houden.

**Replicatie** Eén van de testen is de beschikbaarheidstest. Indien de data maar op een enkele server opgeslagen is, zal de data op de uitgeschakelde server niet langer beschikbaar zijn. Met replicatie zal de data op verschillende servers opgeslagen worden en kan de data in theorie nog beschikbaar zijn in het geval van een enkele uitgeschakelde server.

### 3. IMPLEMENTATIE

---

**Data distributie** Het is de bedoeling om systemen te testen die een grote hoeveelheid data kunnen opslaan. Om aan deze vereiste te voldoen, is het nodig dat elke server niet al de data opslaat bij een grote dataset.

**Ondersteuning voor verschillende query methodes** Bij de testen worden er 5 soorten queries uitgevoerd: invoegen, aanpassen, verwijderen en het opvragen van een individueel of meerdere record. De DBMS moet ondersteuning voor deze queries. De eerste 4 kunnen in al de systemen geïmplementeerd worden met één of meerdere queries. Maar het opvragen van meerdere queries, een scan query, is in bepaalde systemen niet ondersteund. Deze scan query is een query waar het record met een bepaalde sleutel wordt opgevraagd en een aantal records dat hierop volgt, het is *een query met een begin en eind sleutel*.

Voor alle systemen van het boek [37], besproken in sectie A, is het eerste criterium voldaan. Een vergelijking voor de vorige zijn samengevat in tabel 3.1.

Bij de selectie is er naast de 4 criteria, ook gekozen voor systemen van verschillende datamodellen. Samen met mijn collega Arnaud Schoonjans [35], zijn er in 7 verschillende systemen verder onderzocht en als modelsysteem gekozen. Voor deze thesis zijn dit HBase, MongoDB en Pgpool-II verder onderzocht, in de thesis van mijn collega zijn dit Cassandra, Apache CouchDB, Riak en MySQL.

Extra uitleg  
waaro?

## 3.2 Gedetailleerde bespreking van de model DBMS's

In dit gedeelte zal elk model systemen in meer detail uitgelegd worden. Een gemeenschappelijk element bij al deze systemen is dat niet alle instanties dezelfde functie hebben, in andere DBMS's hebben alle instanties dezelfde functie bij het wat de installatie en configuratie kan vereenvoudigen.

Voor elk van de geselecteerde systemen zal de aangeboden API besproken worden met een blik op de datastructuur, daarna zal de systeem architectuur besproken worden.

### 3.2.1 HBase

#### Data structuur[13]

De data in HBase is gesstructureerd in tabellen, bij het aanmaken wordt er een schema voor de tabel gemaakt. Voor elke tabel kunnen de verschillende kolommen meegegeven worden samen met een *kolom familie* voor elke kolom, maar de kolommen kunnen ook gespecificeerd worden bij het schrijven van data. De gegevens per *kolom familie* hebben dezelfde prefix en zullen fysisch samen opgeslagen worden. Indien verschillende kolommen tegelijk worden gelezen of geschreven, is het aangeraden om deze dezelfde *kolom familie* te geven.

### 3.2. Gedetailleerde bespreking van de model DBMS's

---

		Persistentie	Replicatie	Datadistributie	Query soort	
					Aanpassen	Scan
Column	Cassandra	Ja	Master-Master	Ja	Ja	Half
	HBase	Ja	Master-Slave	Ja	Ja	Ja
Document	Apache CoucheDB	Ja	Master-Master	Ja	Nee	Ja
	MongoDB	Ja	Master-Slave	Ja	Ja	Ja
	LightCloud (Tokyo)	Ja	Master-Master	Ja	Nee	Ja
Key-Value	MemcacheDB	Ja	Master-Slave	Nee	Nee	Ja
	Redis	Half	Master-Slave	Nee	Ja	Half
	Riak	Ja	Master-Master	Ja	Nee	Half
	Voldemort	Ja	Master-Master	Ja	Nee	Nee
Relationaleel	MySQL	Ja	Master-Slave	Nee	Ja	Ja
	PostgreSQL	Ja	Master-Slave	Nee	Ja	Ja
	Pgpool-II (PostgreSQL)	Ja	Master-Slave	Ja	Ja	Ja

Tabel 3.1: Ondersteuning van de besproken DBMS's naar de selectie criteria.

Bij *Redis* is er sprake van een snapshot of een log voor de persistentie, de eigenlijke database wordt enkel in het geheugen gehouden. Hierdoor is er maar half sprake , hierdoor kan de database herstelt worden maar is deze niet in het geheugen.

Bij *replicatie* zijn er 2 mogelijke configuraties: master-slave waarbij er verschillende instanties verschillende functies hebben en één de baas is, of master-master waarbij ze allemaal gelijk zijn.

Bij *aanpassen* zijn er systemen die voor een update al de verschillende kolom waarden nodig hebben of maar 1 kolom per waarde ondersteunen.

Bij *scan* is er bij enkele systemen enkel ondersteuning voor het lezen tussen 2 verschillende sleutels. Met het iteratief opvragen van elementen tussen 2 sleutels en het lezen van een beperkte hoeveelheid data, is het mogelijk om een scan query uit te voeren, maar dit is maar halve ondersteuning.

De operaties beschikbaar in dit systeem zijn: get (verkrijgen), put (invoegen), scan en delete (verwijderen). Het aanpassen van gegevens wordt uitgevoerd via een put waarbij een enkele kolom waarde van een record kan aangepast worden. Een scan operatie heeft geen optie om het aantal op te halen records te bepalen. Maar HBase ondersteunt de optie om de batch grootte (bytes) te configureren. Doordat er geweten is hoe groot een individueel record is én hoeveel records er opgevraagd worden, kan de cache grootte zo bepaald worden. Op deze manier is er maar een enkele communicatie met de database nodig is.

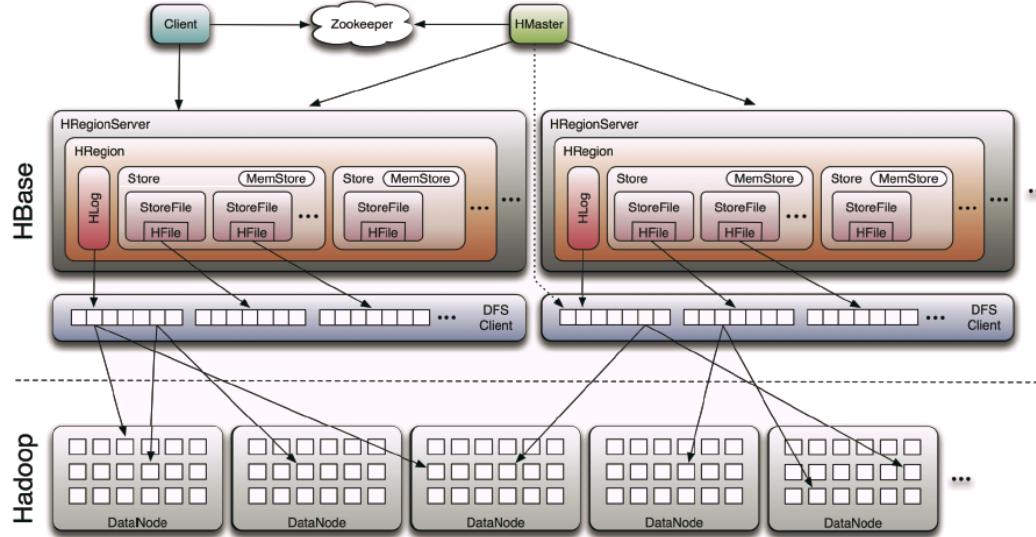
## Architectuur[13]

De gedistribueerde versie van HBase is afhankelijk van 2 andere software systemen: Zookeeper[18] en Hadoop[3]. Hiermee volgt HBase de structuur van Google's BigTable[6] die op zijn beurt afhankelijk is van Chubby[5] en Google File System[14]. Een overzicht van de architectuur bevindt zich in figuur 3.1. De 3 systemen van HBase

### 3. IMPLEMENTATIE

---

zullen kort besproken worden, van HBase naar Zookeeper en Hadoop.



Figuur 3.1: Volledige systeemarchitectuur van HBase met Hadoop en Zookeeper. Bron [19]

**HBase[13]** HBase is een master/slave systeem welke bestaat uit een HMaster en een HRegionServer. De *HMaster* is verbonden met Zookeeper en houdt op deze manier de status en verantwoordelijkheden van de HRegionServers in het oog. Daarnaast is deze ook verantwoordelijk voor het toewijzen van data verantwoordelijkheden. Onder andere wordt de data uit een tabel opgesplitst over verschillende regio's indien een tabel groeit, ook het toewijzen van een HRegion aan een HRegionServer.

De andere soort, een *HRegionServer*, is verantwoordelijke voor de data van een regio en voor het beheren van deze regio's. Een regio is een deel van een tabel met daar in de feitelijke data die opgeslagen is in verschillende Hadoop datanodes. Een HRegionServer zal consistentie en atomaire queries afdwingen in HBase op een enkele record.

**Hadoop[3]** HBase maakt gebruik van het Hadoop Distributed File System (HDFS), een gedistribueerd file systeem ontworpen om te werken op commodity hardware met een hoge fout tolerantie. HDFS heeft een master/slave architectuur en bestaat uit een enkele *namenode*, de master server, die de naamruimte en toegangscontrole onderhoudt, en *datanodes*. De data wordt opgedeeld in blokken die door een verzameling van datanodes worden opgeslagen. Aangezien niet elke node in deze verzameling zit, is er op deze manier data distributie. Deze master/slave configuratie zijn verschillende soorten van services die de administrator afzonderlijk moet opzetten.

In de deze configuratie van HBase, is HDFS de methode om data persistent op te slaan met automatische replicatie en data distributie. Er is ook ondersteuning om de opslag naar Amazon S3 te doen in een gedistribueerde omgeving of deze op de lokale harde schrijf op te slaan bij een configuratie met slechts 1 server.[13]

**Zookeeper[18]** Zookeeper is een service voor het coördineren van gedistribueerde applicatie processen. Deze service biedt primitieven aan om synchronisatie, configuratieonderhoud en benaming te doen. Zookeeper is een gedistribueerd master/slave systeem dat ontworpen is om snel te zijn bij dominantie van leesoperaties.

HBase gebruikt Zookeeper voor het bijhouden van de status van HRegionserver, hun locatie en hun verantwoordelijkheden. De sessie wordt toegekend die een HRegionServer bijvoorbeeld de verantwoordelijkheid voor een Region geeft voor de volgende minuut. Tijdens deze periode kan geen enkele andere HRegionServer een bewerking doen op deze Region, uitgezonderd met de toestemming van de verantwoordelijke server. [13] De duur van een sessie kan geconfigureerd worden in Zookeeper maar wordt in dit geval op de standaard 180 seconden gelaten.

Dit is de globale structuur van het HBase systeem, in het totaal zijn er 5 verschillende soorten services: 2 voor Hadoop, 1 voor Zookeeper en 2 bij HBase. Enkele van deze services worden best gegroepeerd op een enkele instantie: de HDFS namenode, een Zookeeper instantie en de HMaster worden samen op een enkele instantie geplaatst. Hetzelfde geldt voor een datanode en een HRegionServer. Zeker deze laatste heeft een extra performantie invloed: HBase detecteert dat er lokale opslag van de data is en de regio zal steeds deze lokale opslag hebben. Dit zorgt bij leesacties voor een performantie verbetering aangezien de data lokaal gelezen kan worden.

De configuratie van de verschillende systemen gebeurt door middel van configuratiebestanden voor elke service waarna de verschillende systemen zich bij elkaar aanmelden en de volledige configuratie van Region's door het systeem zelf wordt gedaan.

#### 3.2.2 MongoDB[25]

##### Datastructuur

De data in MongoDB is opgeslagen in een database, die op zijn beurt een collectie bevat. Het is niet nodig om een database en collectie op voorhand aan te maken. Beiden worden automatisch aangemaakt bij het wegschrijven van data, indien de collectie nog niet bestaat. Een record is in MongoDB een document en elk record kan verschillende velden hebben. Er zijn uitgebreide query mogelijkheden om data in te voegen, aan te passen, te verwijderen of een scan uit te voeren. Er is ook ondersteuning voor MapReduce[11].

Bij het schrijven van data, kunnen verschillende eisen gesteld worden voor het voltooien van de actie, startende met de actie is over het netwerk verstuurd, de primary heeft de data geschreven tot een meerderheid van de secondaries heeft de data weg geschreven.

Bij het lezen kan men kiezen om de data te lezen van de primary, secondary of de dichtstbijzijnde node. Afhankelijk van de gekozen acties, kan er verondersteld worden dat er een verschillende consistentie garantie zal zijn. Een overzicht van al de

### 3. IMPLEMENTATIE

---

mogelijkheden, kan teruggevonden worden in tabel 3.2. Indien er in de tekst verder niet gespecificeerd wordt welke lees- of schrijfconfiguratie er wordt gebruikt, zijn dit de standaard methodes, respectievelijk primary en normal.

Leesconfiguratie	
Benaming	Omschrijving
Primary	Enkel lezen van de primary
PrimaryPreferred	Lezen van de primary, behalve als de primary onbeschikbaar is, lees dan van secondary.
Secundary	Enkel lezen van een secondary
SecondaryPreferred	Lezen van een secondary, behalve als er geen secondary onbeschikbaar is, lees dan van de primary.
Nearest	Lees van de instantie met de laagste netwerkvertraging, ongeacht het een primary of secondary is.

Schrijf configuratie	
Benaming	Omschrijving
Normal	Wacht tot weggeschreven naar het netwerk socket.
Safe	Wacht op bevestiging van de primary
fsync_safe	Wacht op bevestiging van de primary tot de data is weggeschreven naar harde schijf.
Replica acknowledged	Wacht op bevestiging van primary en één secondary.
Majority	Wacht op bevestiging van meerderheid van de servers

Tabel 3.2: MongoDB: Mogelijke configuraties bij lees- en schrijfbewerkingen

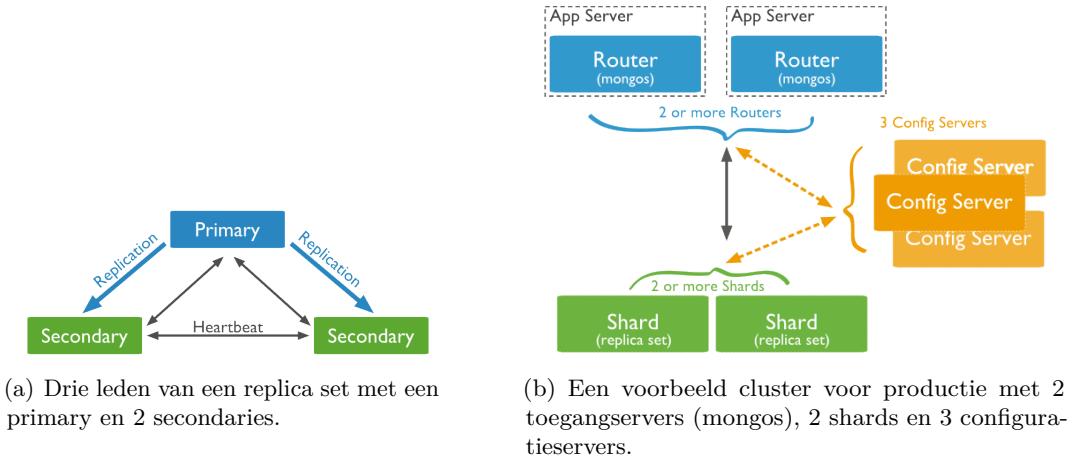
### Architectuur

MongoDB is een DBMS dat de vereisten van replicatie en data distributie op een gelaagde manier tot uitvoering brengt. Eerst zullen instanties gecombineerd worden voor de replicatievereisten invullen, daarna zal horizontale schaalbaarheid ondersteund worden.

**Replicatie[26]** Replicatie gebeurt door middel van een master/slave configuratie tussen verschillende *MongoD* instanties, of in hun termen primary/secondary. Een verzameling van deze MongoD instanties wordt een *replicaset* genoemd. Een replicaset verkiest zelf de primary die verantwoordelijk is voor het afhandelen van de schrijfacties. De data zal vervolgens gerepliceerd worden naar de secondaries. Of deze actie synchroon of asynchroon is, hangt af van de gekozen schrijfconfiguratie. Het is slechts mogelijk om een instantie tot een enkele set toe te voegen. De data is beschikbaar zo lang er meer dan de helft van de servers beschikbaar zijn.

**Data distributie[27]** Horizontale schaalbaarheid wordt in MongoDB bereikt door verschillende replicaset's of zelfstandige MongoD instanties te combineren tot een

### 3.2. Gedetailleerde bespreking van de model DBMS's



Figuur 3.2: MongoDB Architectuur voor replicatie en datadistributie. Bron figuur link: [26], rechts: [27]

cluster. In het geval van een zelfstandige instantie, zal de data niet gerepliceerd worden en wordt om deze reden niet aangeraden voor productie. Voor datadistributie bestaan er 3 verschillende type servers, sharding-, configuratie- en toegangsservers. Een overzicht is gegeven in figuur 3.2(b).

**Shards** De data wordt verdeeld over de verschillende shards nadat is aangegeven dat men deze wilt verdelen over de cluster. Deze verdeling wordt automatisch aangepast indien een enkele shard te groot wordt.

**Configuratie servers** De configuratie servers slaan de meta data van de cluster op zoals de verschillende shards en replicaset's. Deze configuratie set bestaat uit 1 tot 3 servers, voor productie zijn 3 servers aangeraden. Deze servers verdelen de data over de verschillende shards en zullen een de data herstructureren als deze te groot wordt.

**Toegangsserver** De toegangsserver biedt toegang aan tot de cluster en vraagt de configuratie op aan de configuratie servers. Er kunnen een onbepaald aantal toegangsservers zijn in cluster.

De configuratie van de verschillende delen bestaat uit verschillende technieken. Bij replicatie krijgt elke set een naam die in de configuratiebestanden van elke configuratie wordt gezet. Nadien wordt één instantie op de hoogte gebracht van de locatie van de andere instanties. Bij de cluster worden bij het opstarten van de toegangsservers de set van configuratieservers meegegeven, het opzetten van de verschillende shards gebeurt via een toegangsserver m.b.v. de API.

### 3. IMPLEMENTATIE

---

#### 3.2.3 Pgpool-II (PostgreSQL)[29]

Pgpool-II kan op 4 verschillende manieren werken, in deze testen is er gekozen voor de replicatie optie omdat deze zowel replicatie, failover en online recovery aanbiedt en de leesbewerkingen verspreid. Er is de mogelijkheid om ook data distributie aan te bieden maar dit is niet getest. Door de datadistributie bovenop de replicatie te zetten, volgt MongoDB hetzelfde principe als MongoDB.

Datastructuur en de architectuur van Pgpool-II in replicatie mode wordt nu in meer detail besproken.

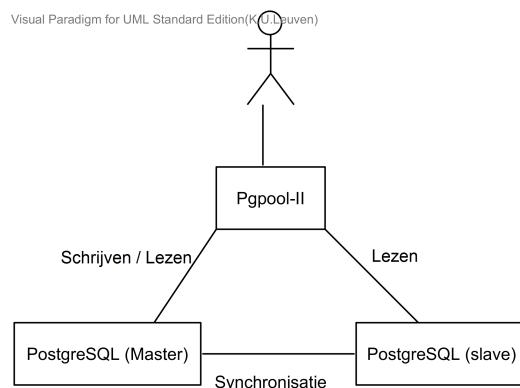
#### Datastructuur

De data structuur en query mogelijkheden van Pgpool-II zijn gelijklopend aan deze van PostgreSQL. Net zoals in PostgreSQL bestaat het systeem uit een schema die verschillende databases kan bevatten. Een database bestaat uit een verzameling van tabellen, een tabel bevat de records. Voor het opslaan van de data dient de volledige tabel met al de kolommen gespecificeerd zijn.

Pgpool-II ondersteunt de volledige query mogelijkheden die in de testen nodig zijn. Er zijn enkele restricties ten opzichte van PostgreSQL die beschreven zijn op in de sectie *Restrictions* van de documentatie[29].

#### Architectuur

Een Pgpool-II infrastructuur bestaat uit 2 delen, een data en routing niveau, een overzicht is gegeven in figuur 3.3.



Figuur 3.3: Systeemarchitectuur van Pgpool-II.

Het data niveau bestaat uit een individuele service uit een PostgreSQL installatie met extra functies, bestanden en een aanpassing aan enkele configuratie bestanden. Daarnaast moet er voor de online recovery ook ssh toegang voorzien worden tussen al de servers. De verschillende data machines hebben een master/slave structuur

waar al de schrijfacties naar de master worden gestuurd en de leesoperaties zijn verdeeld over al de machines. De master doet aan synchronisatie met behulp van de *Write-ahead-log* van PostgreSQL. Dit is een log waar al de verschillende schrijfacties worden opgeslagen. Door een andere server deze te laten uitvoeren is er een gelijke database.

Op routing niveau draait een Pgpool-II service die als management service dient, hij bepaalt wie master en slave is, volgt de status op van de data services en doet aan online recovery. Op het moment dat een databaseverbinding wordt aangemaakt, kiest Pgpool-II naar welke PostgreSQL server dit gaat. Op deze manier wordt de leesbelasting verdeeld.

Pgpool-II kan ook in de parallel mode werken zodat er de mogelijkheid is tot horizontale schaalbaarheid, ook is er de mogelijkheid om caching aan te zetten en een integratie met Memcache is ondersteund.

### 3.3 Selectie en uitwerking van de testsoftware

De testen zijn geïmplementeerd als een uitbreiding van YCSB[8] omwille van verschillende redenen. Allereerst is de broncode publiek beschikbaar onder Apache 2.0, YCSB is een uitgebreid systeem voor het uitvoeren van performantie benchmarking, dit op basis van het meten van de vertraging op een query voor verschillende DBMS's. Hierdoor heeft deze al een uitgebreide ondersteuning voor tal van DBMS's, waaronder al de gekozen systemen. Deze ondersteuning is nog verder geoptimaliseerd voor de gekozen systemen zodat er maximaal gebruik gemaakt wordt van de functionaliteiten van elk systeem. Een concreet voorbeeld: bij het opstellen van de scan queries wordt rekening gehouden met het benodigd aantal records wat standaard in YCSB niet gebeurt bij het uitvoeren op een relationele database.

De 2 testen, beschikbaarheidstest en consistentie test, worden op verschillende manieren geïmplementeerd, naar de leidraad van sectie 2.3.

**Beschikbaarheidstest** De beschikbaarheidstest wordt geïmplementeerd door middel van *event support*, hiermee kan er op vooraf gedefinieerde momenten een bepaald Unix commando uitgevoerd worden. De configuratie gebeurt met behulp van een XML bestand met de parameters van B.1 in bijlage, de output komt in het logbestand met de elementen van tabel B.2 in bijlage.

Met behulp van deze uitbreiding zullen de beschikbaarheidstesten nadien uitgevoerd kunnen worden. Er zal gekeken worden naar de verandering in vertraging op een query waarmee kan bekijken worden of het systeem nog beschikbaar is.

**Consistentie testen** Voor de consistentie testen is er een extra module geïmplementeerd die het gedrag van sectie 2.3 uitvoert. In deze uitwerking leest de

### 3. IMPLEMENTATIE

---

schrijver niet zijn eigen data, al zou dit eenvoudig mee geïmplementeerd kunnen worden. Dit is niet getest omdat het niet nodig was in deze testen. De testen kunnen uitgebreid geconfigureerd worden om enkel te testen wat nodig is: een overzicht van de configuratie parameters is te vinden in tabel B.3. Voor elke uitgevoerde query, wordt een record aangemaakt met de data van tabel B.4 in bijlage.

De code van deze testen is beschikbaar op GitHub onder <https://github.com/thuys/YCSB-Implementation>.

## 3.4 Installatie en opstelling van de DBMS's en YCSB

Het uitvoeren van de testen vereist het installeren van het verschillende instanties en de configuratie van de verschillende DBMS's. Voor het uitvoeren van de verschillende testen is het slechts nodig om het systeem een enkele keer op te zetten. Maar om de testen eenvoudiger te kunnen uitvoeren op verschillende infrastructuren en andere gebruikers de resultaten te laten controleren, is de installatie en configuratie van het systeem geautomatiseerd.

De automatisatie gebeurt met het Integrated configuration Management Platform (IMP) beschreven in [40]. Dit modulair framework is uitgebreid met de 3 DBMS's en YCSB waardoor de configuratie als een declaratief gewenste staat wordt uitgedrukt. IMP zal deze staat toepassen op de verschillende systemen bij het uitrollen.

Een uitgebreider bespreking van de uitwerking in IMP kan gevonden worden in bijlage D met het domeindiagram van het systeem, uitleg en voorbeeldcode.

Voor de uitvoering van de testen, is er voor elk DBMS gekozen voor een minimaal aantal instantie dat datadistributie én replicatie ondersteunt. Voor de laatste eigenschap zou de data beschikbaar moeten blijven bij het uitvallen van 1 server na de overgangsperiode. In de testen is er enkel gefocust op het uitvallen van dataservers, niet naar configuratieservers. De configuratieservers zijn enkel minimaal opgezet omdat deze online blijven tijdens de testen.

De opstelling van de systemen is getoond in figuur 3.4, elke uitrol van de systemen zal in meer detail besproken worden nadat de testinfrastructuur is besproken.

De testinfrastructuur is een IaaS (Infrastructure as a Service) gebaseerd op OpenStack<sup>1</sup>. De infrastructuur bestaat uit 3 Dell R610 en R620 servers met een totaal van 196GB RAM, 44 fysische CPU's (88 met hypertreading), verbonden met een Gigabit switch. Deze infrastructuur is gedeeld met andere gebruikers. Elke instantie heeft 2 virtuele CPU's, 4GB RAM en 50GB schijf ruimte. De instanties worden verdeeld over de verschillende servers. De netwerkinfrastructuur heeft een gemiddelde ping van 0.4ms naar elke node ( $\sigma = 0.2$  bij 10 000 ping's).

---

<sup>1</sup><https://www.openstack.org/>

**HBase** Voor HBase wordt de data standaard 3 maal gerepliceerd en zijn er voor datadistributie dus 4 data instanties nodig. Elk van deze instanties hebben een HBaseRegionServer en Hadoop datanode. Daarnaast zijn er nog een HMaster, Zookeeper en Hadoop namenode nodig die samen op een enkele instantie worden uitgerold. In het totaal zijn er 5 instanties. Een overzicht van de infrastructuur getoond in figuur 3.4(a). De installatie en configuratiebestanden kunnen gevonden worden op <https://github.com/thuys/hbase>.

**Pgpool-II** Bij Pgpool-II is er ondersteuning voor horizontale schaalbaarheid in de parallel mode maar dit is niet getest. Om deze reden is er enkel replicatie toegepast waarvoor er 3 instanties zijn: een Pgpool-II instantie en twee PostgreSQL instanties. De configuratie van deze instanties zijn standaard met uitzondering van de activatie van de Write-Ahead-Log van PostgreSQL en de activatie van de replicatie mode in Pgpool-II. Een overzicht van de infrastructuur is getoond in figuur 3.4(b). De installatie en configuratiebestanden kunnen gevonden worden op <https://github.com/thuys/postgresql>.

**MongoDB** MongoDB heeft ondersteuning in replicatie en datadistributie. Voor het beschikbaar zijn van de data bij het uitzetten van een enkele instantie, zijn er 3 MongoDB datanodes nodig in een replicaset. De data wordt verdeeld over 2 replicaset met behulp van sharding op basis van de hash van de key voor het opzoeken van de query. Omdat de toegangsserver en configuratie instanties niet veel resources innemen, zijn deze verspreid over de verschillende data instanties. Er zijn meerdere toegangsnodes geplaatst om de queries te verdelen naar verschillende toegangsnodes, bij de beschikbaarheidstesten zal een toegangsnode altijd beschikbaar blijven. In het totaal zijn er 6 instanties nodig. Deze zijn beschreven in 3.4(c) in bijlage. De installatie en configuratiebestanden kunnen gevonden worden op <https://github.com/thuys/mongodb>.

**YCSB** YCSB kan naar meerdere instanties uitgerold worden. In deze testen is er gekozen om maar een enkele instantie uit te rollen om de testen eenvoudiger te kunnen uitvoeren. Een overzicht is getoond in figuur 3.4(d).

## 3.5 Uitvoeren van de calibratie en testen

Voor het uitvoeren van de volledige benchmarking dient eerst de verdeling van de type queries gespecificeerd worden, deze zijn voor alle verschillende systemen gelijk. Een overzicht van deze parameters kunnen gevonden worden in tabel 3.3. 40% van de uitgevoerde queries past de database aan, er is dus een dynamische database. Bij het lezen wordt er de helft van de keren in batch gelezen met gemiddeld 50 records per bewerking. Tenslotte wordt er met een *zipfian* verdeling gekozen om regelmatig dezelfde records te lezen waardoor de data aan het DBMS uit cache gelezen kan worden.

### 3. IMPLEMENTATIE

---

Voor later de data optimaal te kunnen analyseren, wordt er elke second de gemiddelde vertraging gelogd voor elk type query.

**Calibratie testen** Voor de calibratie van de omgeving zijn er 2 soorten testen gedraaid, de parameters voor het aantal connecties kunnen gevonden worden in tabel 3.4. De parameters voor het aantal queries per second zijn te vinden in tabel 3.5. In dit geval is het aantal gebruikers bepaalt door de uitkomst van de calibratie van het aantal gebruikers.

Naam	Waarde
Aantal velden	10 (1 key veld)
Record grootte	1KB (100byte/veld)
Lees alle velden	true
Invoeg queries ( <i>insert</i> )	20%
Lees queries ( <i>select</i> )	40%
Aanpas queries ( <i>update</i> )	20%
Scan queries ( <i>scan</i> )	20%
Opvraag verdeling	zipfian ( <i>bepaalde records worden veel gelezen, andere weinig</i> )
Maximale scan grootte	100
Verdeling scan grootte	uniform

Tabel 3.3: Overzicht van de query parameters

Naam	Waarde
Ingeladen records	300 000
Pauze	50s
Executie tijd	600s
Aantal gebruikers	1, 2, 3, 4, 5, 7, 10, 15, 20, 30, 40, 50, 75, 100

Tabel 3.4: Calibratie: Overzicht van de parameters voor het testen van het aantal gebruikers

Naam	Waarde
Ingeladen records	300 000
Pauze	50s
Executie tijd	600s
Theoretisch aantal records per seconde	20, 50, 100, 150, 200, 250, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 2500, 3000

Tabel 3.5: Calibratie: Overzicht van de parameters voor het testen van het aantal records per seconde

**Beschikbaarheidstesten** Bij het uitvoeren van de testen op de beschikbaarheid van de verschillende systemen zijn de parameters in tabel 3.6 gebruikt. Het aantal

vooraf ingeladen records is op 300 000 geplaatst zodat zowel HBase als MongoDB aan sharding doen. De commando's voor het stoppen en starten van de systemen zijn te vinden in tabel B.5. Voor Pgpool-II is er een extra commando toegevoegd dat na het herstarten van de systemen wordt uitgevoerd. Dit komt omdat er geen automatische recovery in Pgpool-II is. Tenslotte worden deze testen uitgevoerd op al de data nodes, een overzicht hiervan met de overeenkomstige service is te vinden in tabel 3.7. De testen voor MongoDB zijn uitgevoerd op replicaset 2, dit is de set zonder enige configuratie- en toegangsservers servers. Een enkele replicaset is voldoende omdat de verschillende replicasetten dezelfde functie hebben, enkel andere data opslaan.

Naam	Waarde
Ingeladen records	300 000
Pauze	50s
Executie tijd	900s
Opstart kost	100s
Stoppen	Op 300s
Starten	Op 600s

Tabel 3.6: Beschikbaarheidstesten: Overzicht van de parameters

Naam	Instanties	Service naam
HBase	HB2, HB3, HB4, HB5	hbase-regionserver
MongoDB	MDB4, MDB5, MDB6,	mongodb-dataserver
Pgpool-II	PG1, PG2	postgresql

Tabel 3.7: Beschikbaarheidstesten: Overzicht van de instanties naar figuur 3.4

**Consistentie testen** Voor de consistentie testen moeten de parameters van tabel B.3 geconfigureerd worden, de parameters zijn te vinden in tabel 3.8. Deze test wordt uitgevoerd op HBase en MongoDB.

Om de analyse van de gegevens eenvoudiger te maken is er bij MongoDB gekozen om de test enkel uit te voeren op een replicaset en niet op een volledige cluster. Er is de aanname dat het consistentieverster afhankelijk is van de lengte tot de gegevens beschikbaar zijn op al de verschillende instanties van een replicaset. Het testen van een cluster voegt zo extra complexiteit toe. Deze test zou in de toekomst ook uitgevoerd kunnen worden op een cluster maar is in dit geval niet gedaan.

## 3.6 Verzamelen en analyse van de testresultaten

De analyse van de data gebeurt aan de hand van de informatie die gelogd wordt tijdens de executie van de testen. Voor alle mogelijke testen maakt R-code de data visueel in verschillende grafieken. Voor elke test kan de data op een andere wijze voorgesteld worden.

### 3. IMPLEMENTATIE

---

Naam	Waarde	
	HBase	MongoDB
Ingeladen records	300 000	
Pauze	50s	
Executie tijd	900s	
starttime	30s	
readThreads	10	5
consistencyDelayMillis	30ms	10ms
newrequestperiodMillis		500ms
readProportionConsistencyCheck		50%
updateProportionConsistencyCheck		50%
stopOnFirstConsistency		True
maxDelayConsistencyBeforeDropInMicros		300ms
timeoutConsistencyBeforeDropInMicro		300ms

Tabel 3.8: Consistentie testen: Overzicht van de parameters

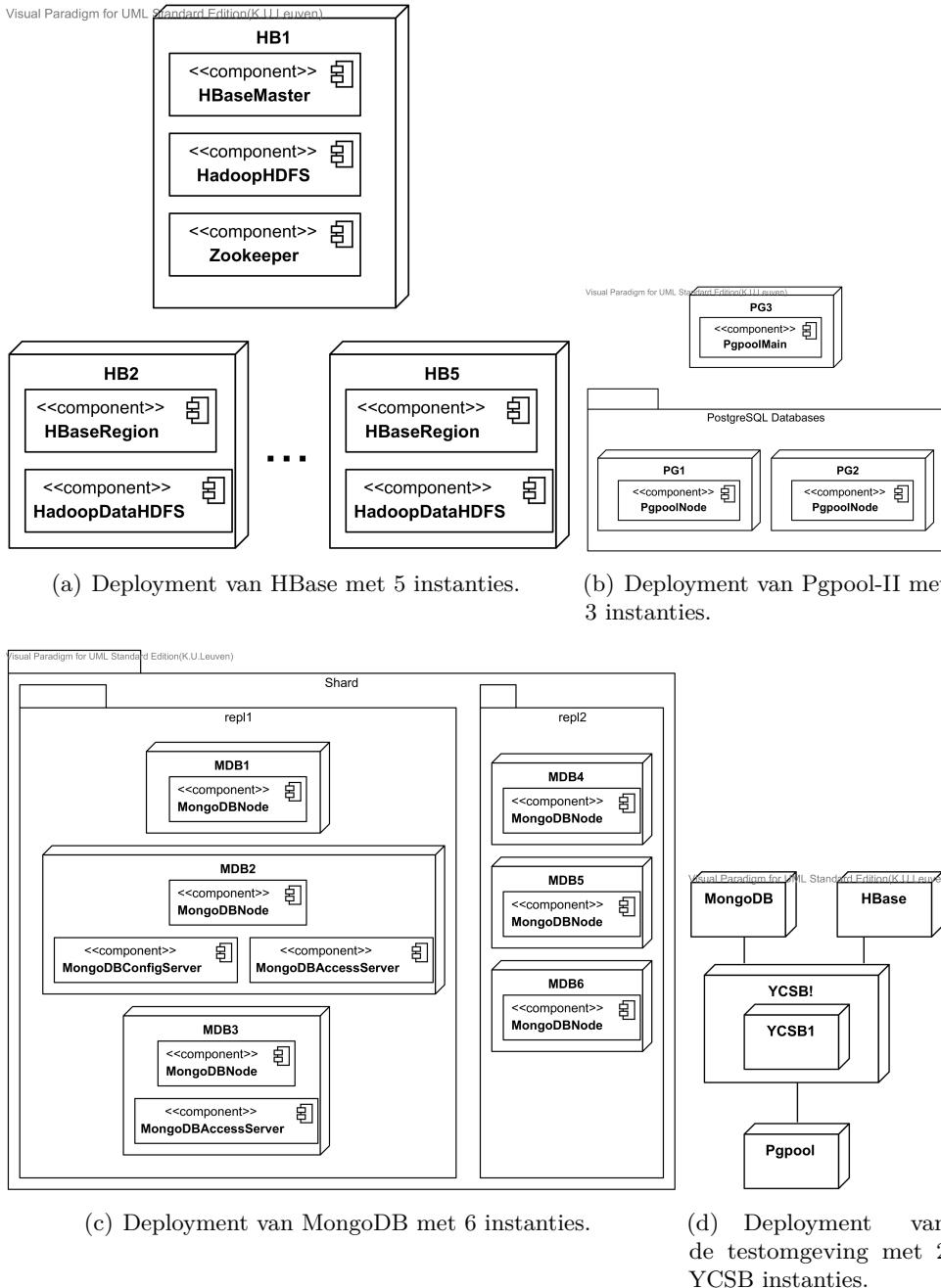
De uitleg en voorbeelden van deze grafieken zullen getoond worden bij het presenteren van de resultaten in het volgende hoofdstuk.

De R code kan gevonden worden op GitHub <https://github.com/thuys/YCSB-R-Scripts>.

## 3.7 Conclusie

In dit hoofdstuk is de vertaling gemaakt van een theoretisch testmodel tot de implementatie. Daarbij zijn keuzes gemaakt en is de configuratie voor de volgende testen vastgelegd. De code voor elk gedeelte is te vinden op Github zodat anderen de testen kunnen reproduceren en aanpassen. De installatie van de model DBMS's is geautomatiseerd zodat deze met slechts weinig kennis kunnen opgesteld worden. De testresultaten zullen tenslotte visueel worden voorgesteld zodat het eenvoudiger is om de data te verwerken.

De grote lijnen van de testmethode uit hoofdstuk 2 zijn geïmplementeerd, maar bepaalde mogelijkheden zijn niet geïmplementeerd. Dit is onder meer het geval voor lezen na het schrijven in de consistentie test en controleren of een waarde beschikbaar is in andere instanties na het platleggen van een instantie in de beschikbaarheidstest.



Figuur 3.4: Deployment van de verschillende DBMS's en de testomgeving.



# Hoofdstuk 4

## Observaties

Dit hoofdstuk behandelt de resultaten van de consistentie- en beschikbaarheidstesten uitgevoerd op HBase, MongoDB en Pgpool-II. In dit hoofdstuk zal er geen verklaring en conclusies getrokken worden over de testen, dit gebeurt in het volgende hoofdstuk.

De testen zullen zullen besproken worden in 3 delen. Eerst zullen de resultaten van de calibratie getoond worden met een selectie van het aantal gebruikers en bewerkingen per seconde, waarmee de beschikbaarheids- en consistentietesten uitgevoerd worden. Vervolgens zullen de beschikbaarheidstesten aan bod komen en tenslotte de consistentietesten.

De ruwe testdata is beschikbaar op <https://github.com/thuys/YCSB-Testdata>.

### 4.1 Calibratie

**Aantal gebruikers** De resultaten van de calibratietest voor het aantal gebruikers kunnen gevonden worden in figuur 4.1.

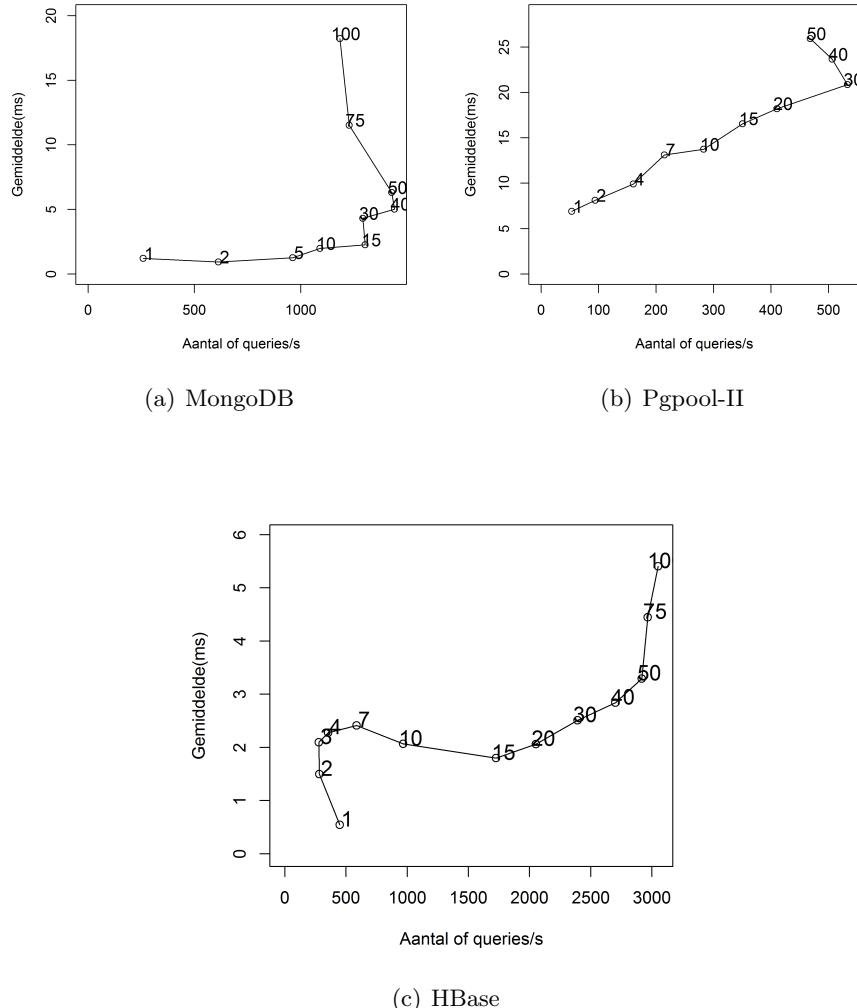
Het aantal gebruikers wordt zo gekozen dat het totale aantal queries zakt of voor een sterke groei in vertraging zorgt, dit zorgt voor de gegevens in tabel 4.1. Bij MongoDB is er voor een lage waarde van 15 gebruikers gekozen, in plaats van 50. De reden hiervoor is dat de variatie in de vertraging groter wordt bij meer gebruikers, wat de vergelijking van de vertraging in de overige testen potentieel moeilijker maakt.

DBMS	Aantal gebruikers
HBase	50
MongoDB	15
Pgpool-II	30

Tabel 4.1: Calibratie: Aantal gebruikers per test voor de verschillende DBMS's

#### 4. OBSERVATIES

---



Figuur 4.1: Calibratie: Overzicht van het aantal queries tot de gemiddelde vertraging voor verschillend aantal gebruikers. Elk datapunt stelt een verschillend aantal gebruikers voor met het aantal rechtsboven het punt.

Op de x-as is het gemiddeld aantal queries per seconden getoond over een periode van 600s, op de y-as de gemiddelde vertraging. De verschillende punten stellen een aantal gebruikers voor die zijn aangegeven met het bijhorend getal.

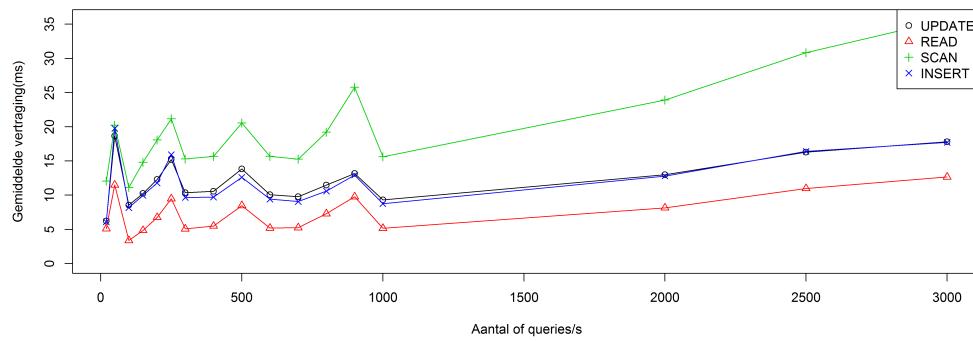
**Aantal queries per seconde** De resultaten voor de calibratietest voor het aantal queries per seconden kunnen gevonden worden in de figuren 4.2, 4.3 en 4.4 voor respectievelijk HBase, MongoDB en Pgpool-II. Deze figuren tonen in de bovenste figuur de gemiddelde vertraging op een query afhankelijk van het aantal queries per seconde. Naarmate het aantal queries toeneemt stijgt de vertraging, uitgezonderd bij laag aantal queries. De onderste figuur toont op de y-as de verhouding tussen het

#### 4.1. Calibratie

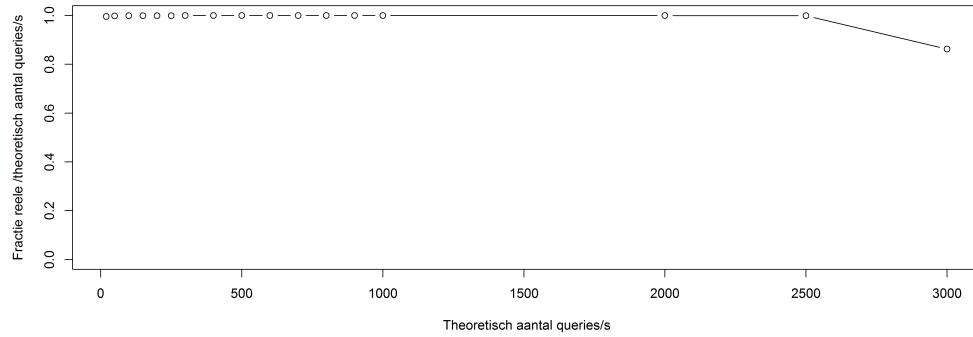
---

eigenlijk aantal uitgevoerde queries per seconde t.o.v. het gevraagde aantal queries per seconde. Een fictief voorbeeld: bij het vragen van 100 queries/sec worden er in de praktijk maar 60 uitgevoerd, dit zorgt voor een waarde van 0.6.

Met beide figuren samen, kan een matige belasting gekozen. Een matige belasting is een belasting waarbij de tweede figuur voor elk systeem de waarde 1 zo dicht mogelijk benaderd en de vertraging nog niet te veel is gestegen t.o.v. van een lage belasting. De gekozen waarde zijn te vinden in tabel 4.2. De testen zouden ook uitgevoerd kunnen worden onder een andere belasting en de resultaten zouden vergeleken kunnen worden.



(a) Gemiddelde vertraging bij een verandering in aantal queries per seconde.

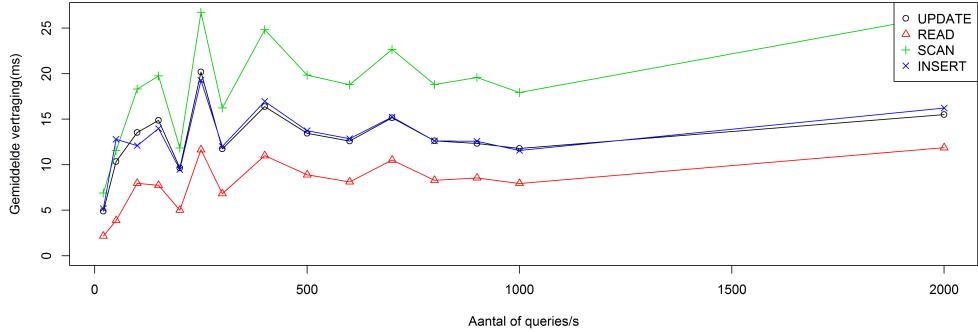


(b) Verhouding van het aantal uitgevoerde queries ten opzichte van het aantal gevraagde queries.

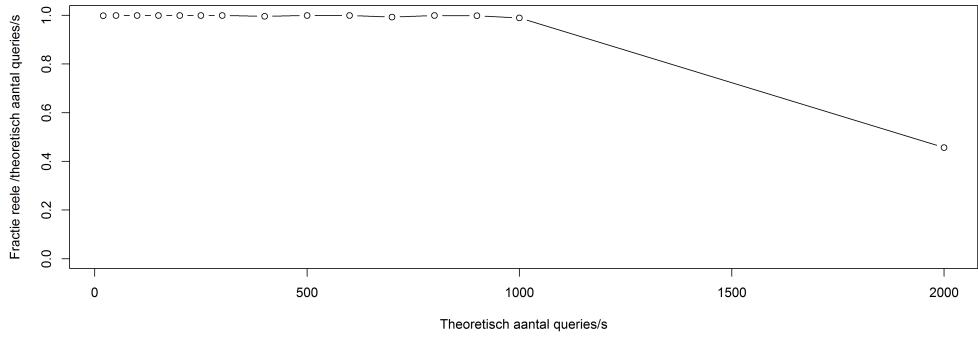
Figuur 4.2: Calibratie: Overzicht van de vertraging t.o.v. het theoretisch aantal aanvragen met een vergelijking hoeveel werkelijke aanvragen er waren voor HBase.

#### 4. OBSERVATIES

---



(a) Gemiddelde vertraging bij een verandering in aantal queries per seconde.



(b) Verhouding van het aantal uitgevoerde queries ten opzichte van het aantal gevraagde queries.

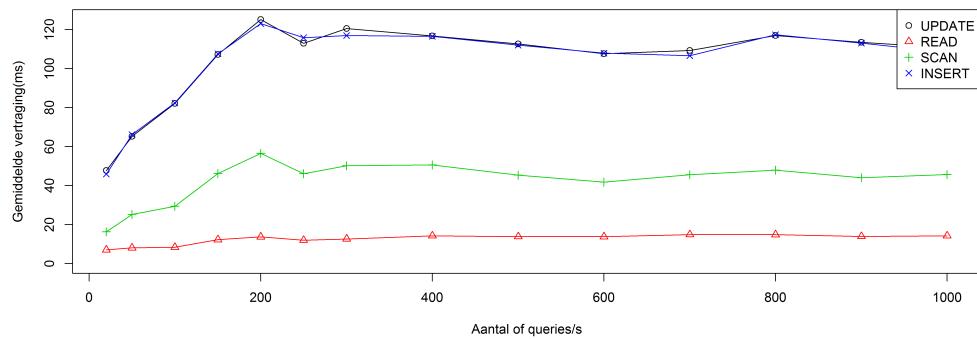
Figuur 4.3: Calibratie: Overzicht van de vertraging t.o.v. het theoretisch aantal aanvragen met een vergelijking hoeveel werkelijke aanvragen er waren voor MongoDB.

DBMS	Aantal requests per seconde
HBase	600
MongoDB	200
Pgpool-II	100

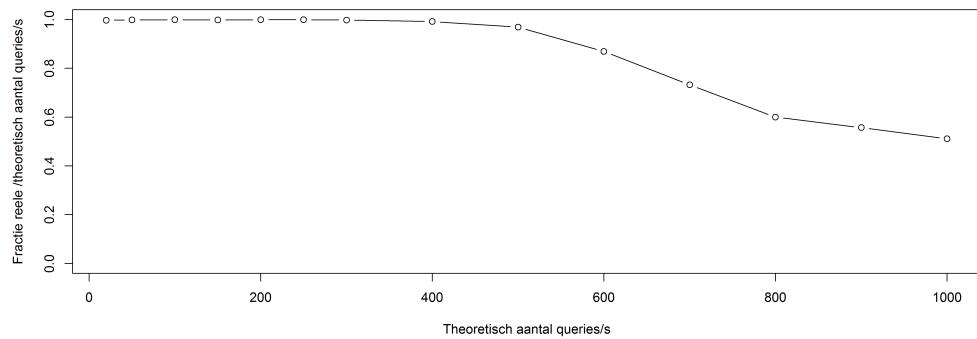
Tabel 4.2: Calibratie: Aantal queries per seconde per test bij een matige belasting voor de verschillende DBMS's.

#### 4.1. Calibratie

---



(a) Gemiddelde vertraging bij een verandering in aantal queries per seconde.



(b) Verhouding van het aantal uitgevoerde queries ten opzichte van het aantal gevraagde queries.

Figuur 4.4: Calibratie: Overzicht van de vertraging t.o.v. het theoretisch aantal aanvragen met een vergelijking hoeveel werkelijke aanvragen er waren voor Pgpool-II.

## 4. OBSERVATIES

---

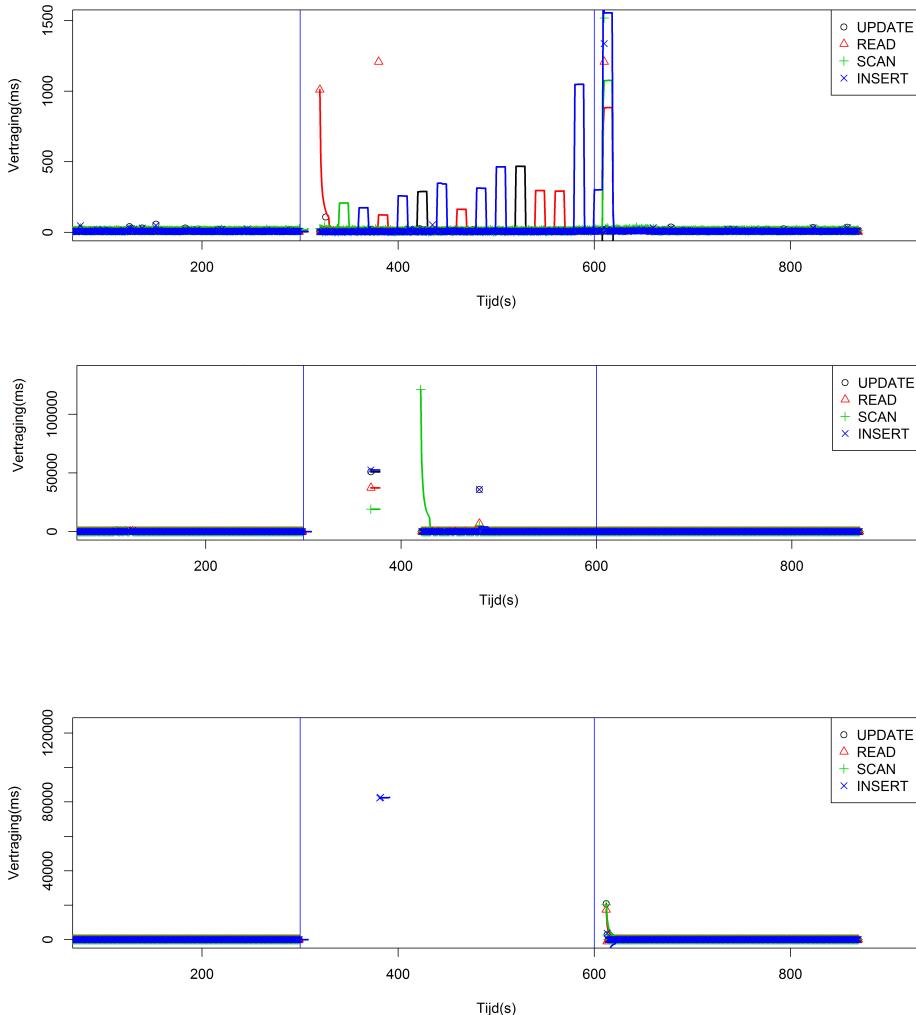
### 4.2 Beschikbaarheidstest

Bij de beschikbaarheidstesten kunnen de gegevens op verschillende manieren voorgesteld worden: de vertraging per query over de hele test, de vertraging tijdens het stoppen en starten van systemen of een vergelijking van de vertraging voor het stoppen (150-250s), tussen het stoppen en starten(400-500s) en na het herstarten (700-800s).

Voor elk systeem zullen enkele grafieken getoond worden, de rest zal besproken worden in de tekst bij de figuren. De grafieken van al de testen zijn beschikbaar op de link gegeven in het begin van het hoofdstuk.

Een punt op de grafiek stelt de gemiddelde vertraging van 1 seconde voor, de lijn het gemiddelde over 10 seconden.

## 4.2. Beschikbaarheidstest



Figuur 4.5: **Beschikbaarheid van HBase**

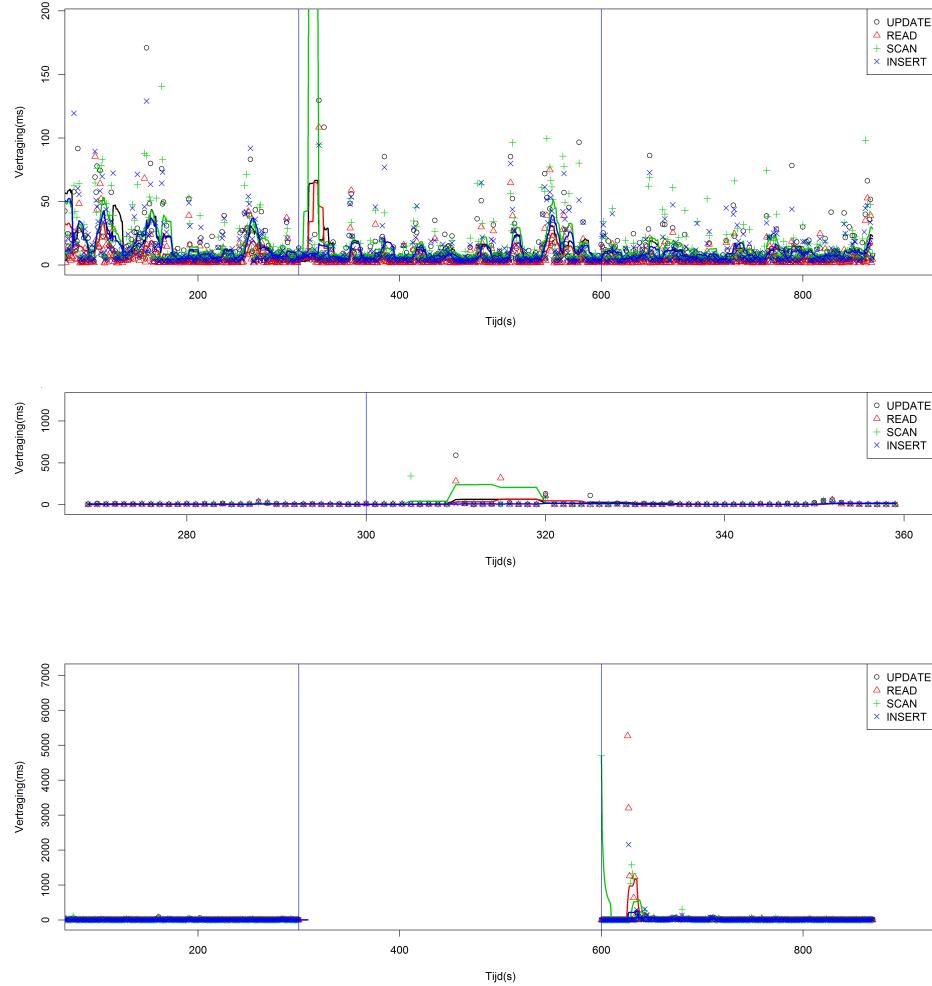
Bij HBase zijn er verschillende reacties op het stopzetten van een node. Bij het zacht stoppen van een instantie, is er een onderbreking van ongeveer 20 seconde in de testen. Daarna kunnen er nog verhogingen in de queries af en toe optreden. Een voorbeeld, zie de eerste figuur.

Bij de netwerk onderbreking is er in de testen een onderbreking van gemiddeld ongeveer 100 seconden. Daarna is het terug stabiel. Zie de middelste figuur.

Bij een harde stop is er een combinatie van de netwerk onderbreking én is het af en toe zo dat de volledige periode geen queries mogelijk zijn. Zie de laatste 2 figuren. Tijdens de onderbreking (400-500s), is er geen significante verandering in de vertraging van de uitgevoerde queries gemeten (t.o.v. 150s - 250s en 700s - 800s).

#### 4. OBSERVATIES

---

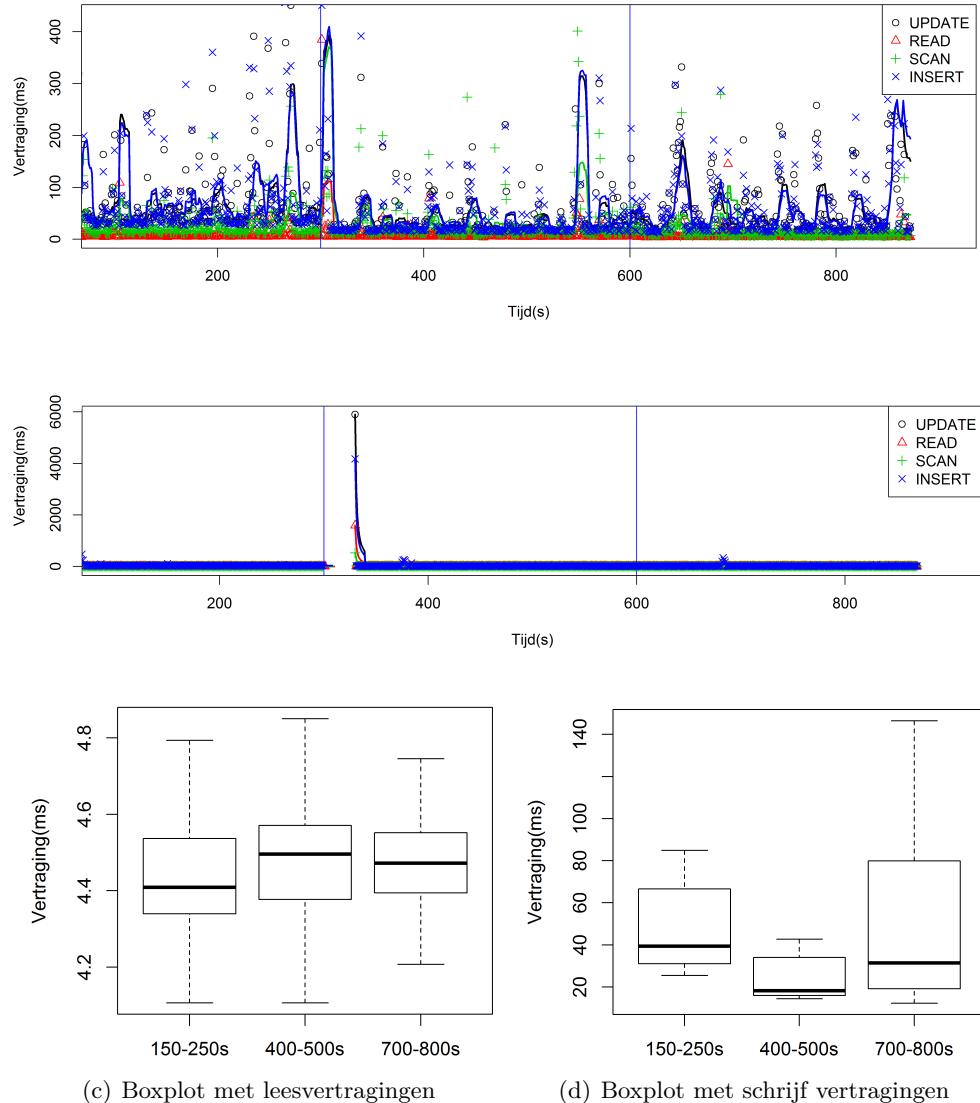


Figuur 4.6: **Beschikbaarheid van MongoDB**

MongoDB heeft verschillende reacties op het stopzetten bij standaard configuratie van de queries (normaal schrijven en lezen van de primary). In het geval van zacht of hard stoppen is er geen verschil in de reactie, bij 2/3 van de keren is er geen verschil merkbaar, bij 1/3 van de keren is er tijdelijke verhoging van de vertraging, een voorbeeld toont dat de scan operatie voor 2 seconden uitgesteld wordt. De eerste figuur is een overzicht, de middelste voor een zoom naar de stop.

Bij het onderbreken van het netwerk is er in bepaalde geen significante verandering, op andere momenten is een gedrag soortgelijk aan dat bij een zachte stop te merken. In andere gevallen is het zo dat er geen queries mogelijk zijn gedurende de volledige netwerk onderbreking. Zie de derde figuur.

Tijdens de onderbreking (400s- 500s), is er geen significante verandering in de vertraging van de uitgevoerde queries gemeten (t.o.v. 150-250s en 700s - 800s).


 Figuur 4.7: **Beschikbaarheid van Pgpool-II**

Pgpool-II is een geen verschil tussen een harde of zacht stop. In beide gevallen is er tijdelijk een onderbreking van al de queries, er is een verhoogde vertraging van ongeveer 2 seconden, een voorbeeld bevindt zich in de bovenste figuur.

Bij een netwerk onderbreking, zijn er enige tijd geen queries mogelijk en na 30 seconden was de onderbreking over in de testen. Een voorbeeld bevindt zich in de middelste figuur.

Tijdens de onderbreking is er een verandering naar schrijfbewerkingen toe: deze nemen significant minder tijd in beslag. Dit geldt niet voor leesbewerkingen. Voorbeelden uit de testen bevinden zich voor beiden in figuren (c) en (d).

Het herstel van een server na deze opnieuw online gebracht hebben, lukt niet tijdens de testen. Enkel als alle connecties zijn verbroken na de test, slaagt het herstel.

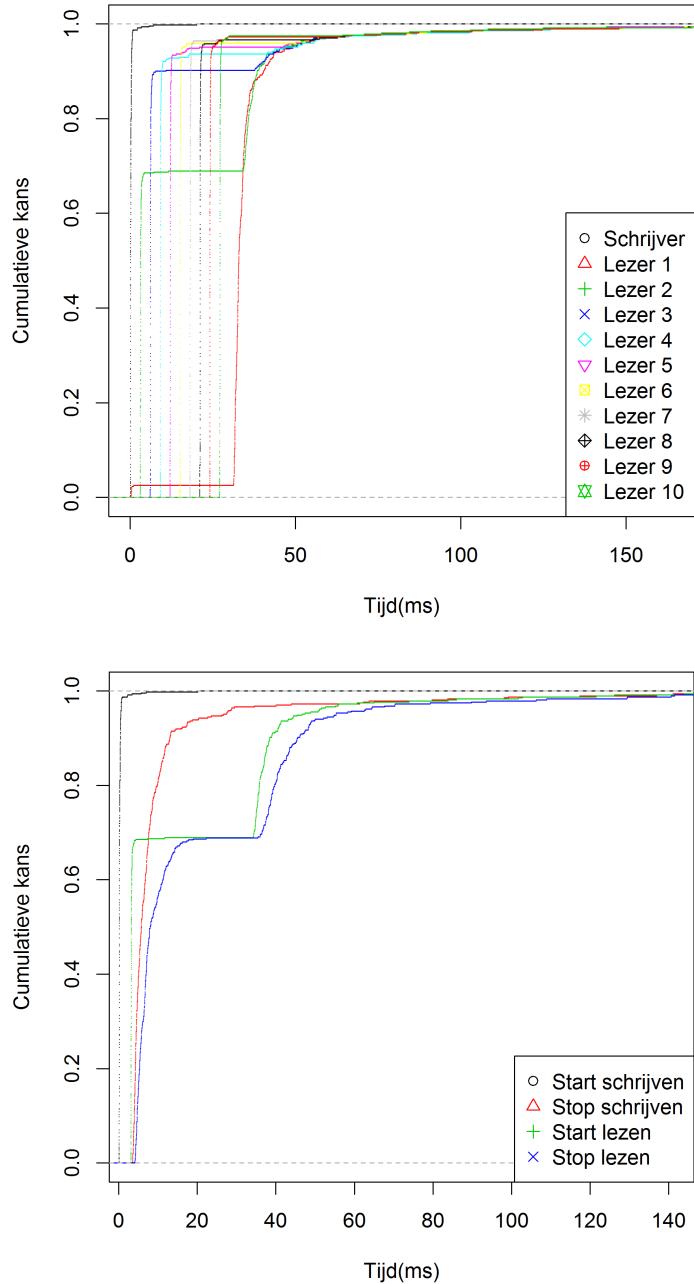
#### 4. OBSERVATIES

---

### 4.3 Consistentietest

Voor de consistentietesten worden er empirische verdelingsfuncties gebruikt. Dit zijn functies waarbij op de y-as het percentage staat van de waarden kleiner dan x.

Voor de consistentie testen wordt er op de x-as de start- en/of stoptijdstippen van de verschillende soorten getoond. Het verschil tussen de y-waarde van de start- en stoptijdstippen geeft aan hoeveel queries er op dat moment uitgevoerd worden. De getoonde tijdstippen van een lezer zijn de eerste keer dat deze de correcte data leest.

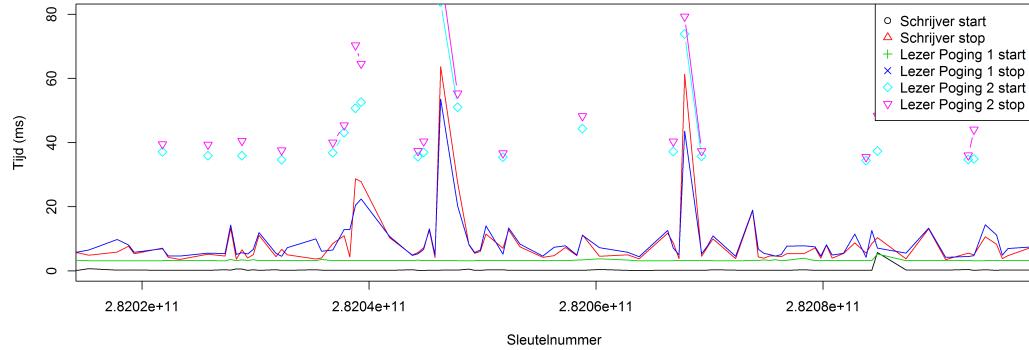
Figuur 4.8: **Consistentie van HBase**

HBase heeft geen verschil tussen het invoegen of aanpassen van data naar consistentie, dit zijn dezelfde queries. Daarnaast is de enige configuratie mogelijkheid voor het lezen of schrijven het in- of uitschakelen van de caches aan de gebruikerskant. In al de testen zijn deze uitgeschakeld.

Figuur (a) toont een overzicht van de verschillende starttijdstippen voor het lezen van consistentie data. Figuur (b) toont de start- en eindtijdstippen voor lezer 2 naast deze voor de schrijver. Lezer 2 start met op 3ms en leest vervolgens elke 30ms tot de nieuwe waarde is gelezen. De maximale waarde van de x-as is zo gekozen dat voor elke dataset minstens 99% van de data getoond is.

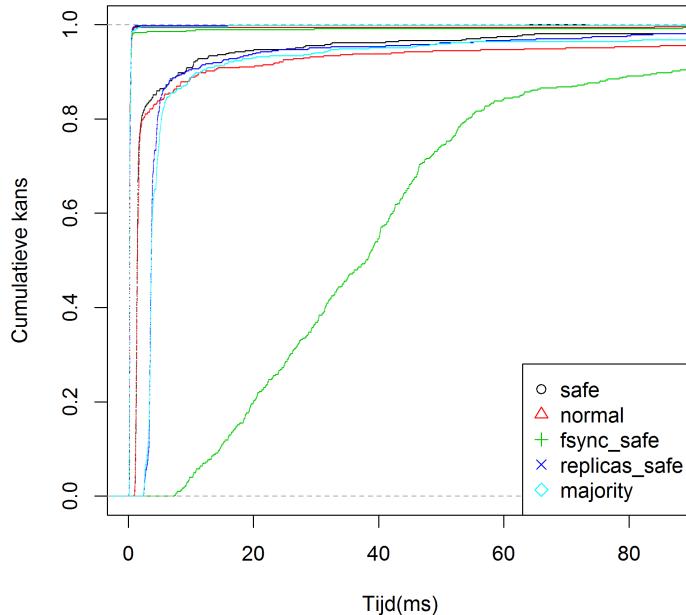
#### 4. OBSERVATIES

---



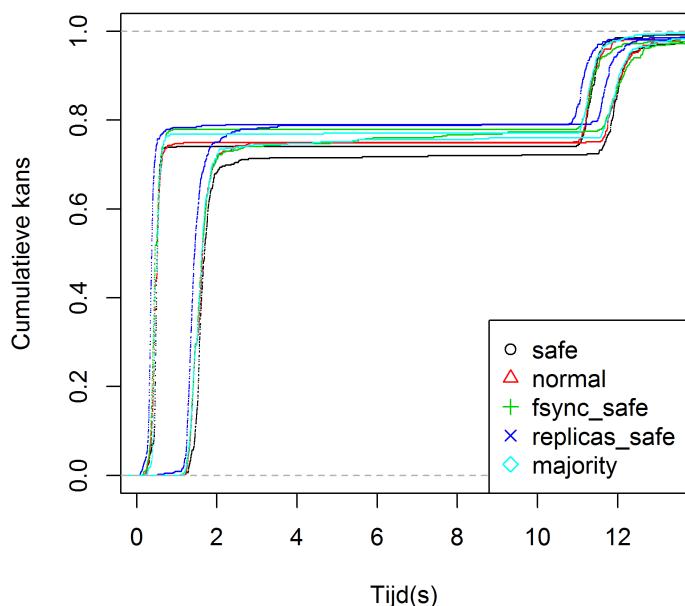
Figuur 4.9: **Consistentie van HBase: Tijdsverloop**

In de figuur zijn op de x-as verschillende pogingen doorheen de tijd voorgesteld. Op de y-as staat de vertraging voor elke bewerking. De lezer zal een tweede keer moeten lezen de eerste poging van de leesactie gedaan is voor de schrijfactie voltooid is. Ook volgt het einde van de leesactie het einde van de schrijfactie.



Figuur 4.10: **Consistentie van MongoDB: Verloop van schrijfoperaties**

Dit is de ruwe data van MongoDB's verschillende schrijfoperaties met een 90-percentiel, de start- en stoptijden zijn in dezelfde kleur.

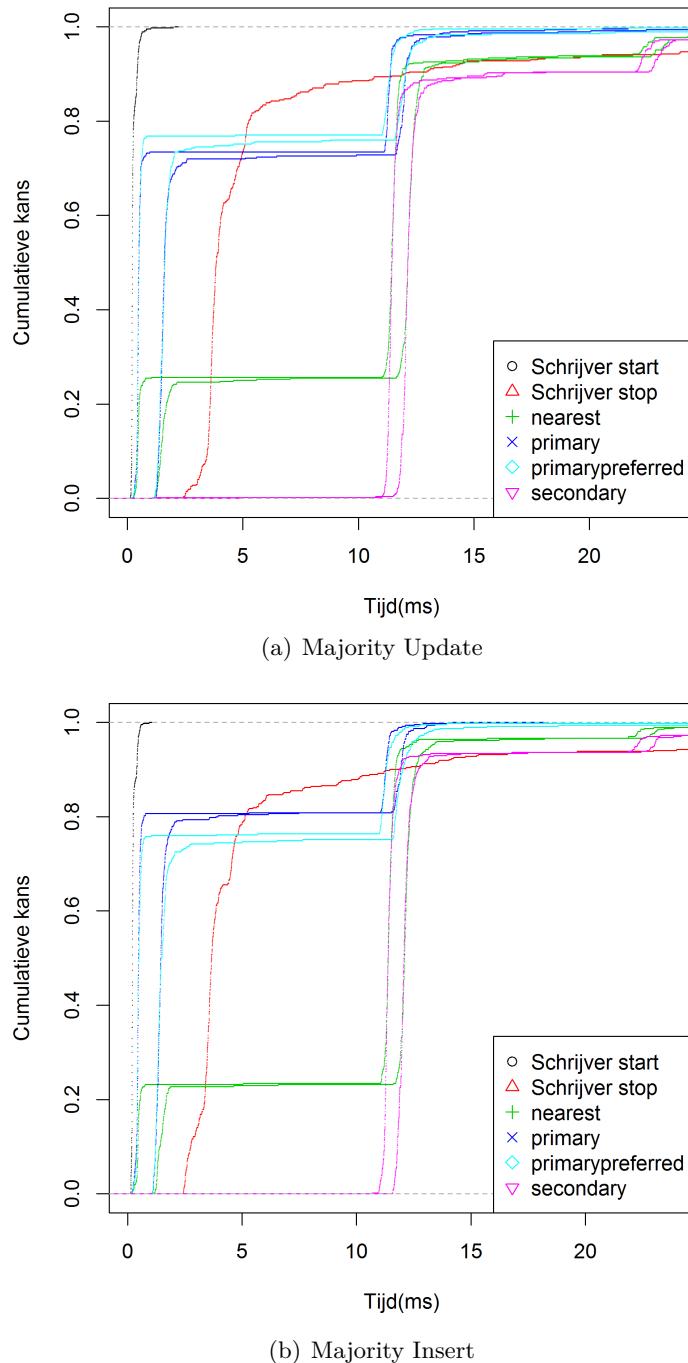


Figuur 4.11: **Consistentie van MongoDB: Leesgedrag bij verschillende schrijfconfiguraties**

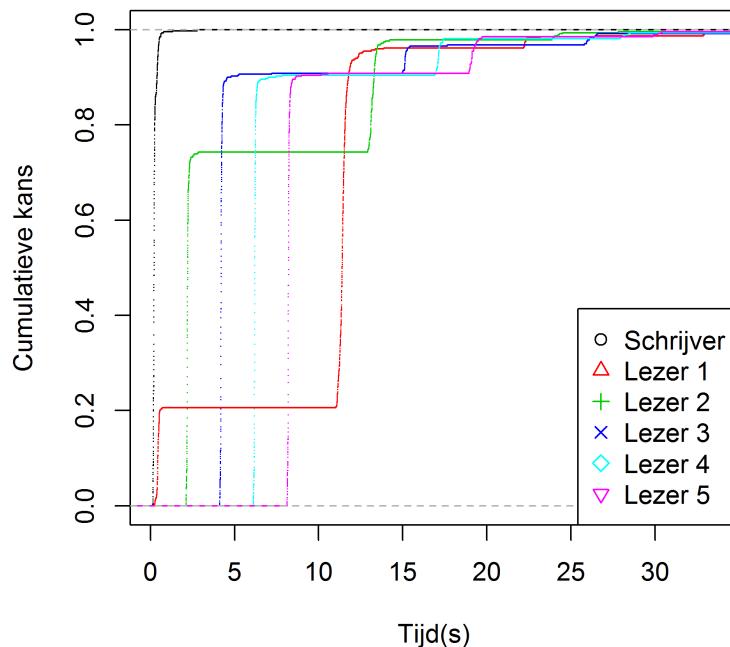
Een vergelijking van het leesgedrag onder verschillende schrijfconfiguratie toont dat er geen significant verschil is. In de figuur is een 99-percentiel getoond voor een update operatie met als leesactie primary, de start- en stoptijden zijn in dezelfde kleur.

#### 4. OBSERVATIES

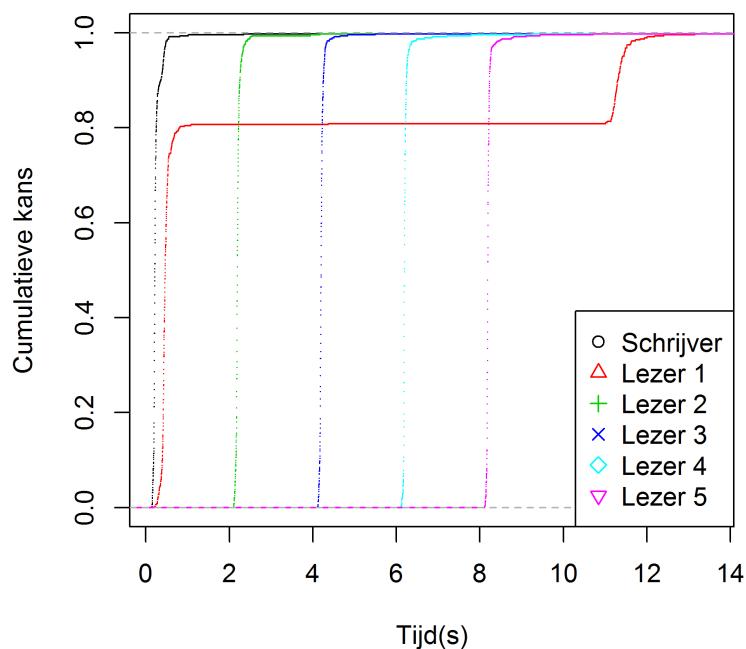
---



**Figuur 4.12: Consistentie van MongoDB: Insert vs Update voor lezer 2**  
 Overzicht van MongoDB's insert vs update met een 99-percentiel voor lezer 2. De start- en stoptijden zijn in dezelfde kleur. Lezer 2 start met lezen op 2ms en vervolgens elke 10ms. Er is geen significant verschil tussen een insert of een update bij dezelfde leesconfiguratie en dezelfde lezer.



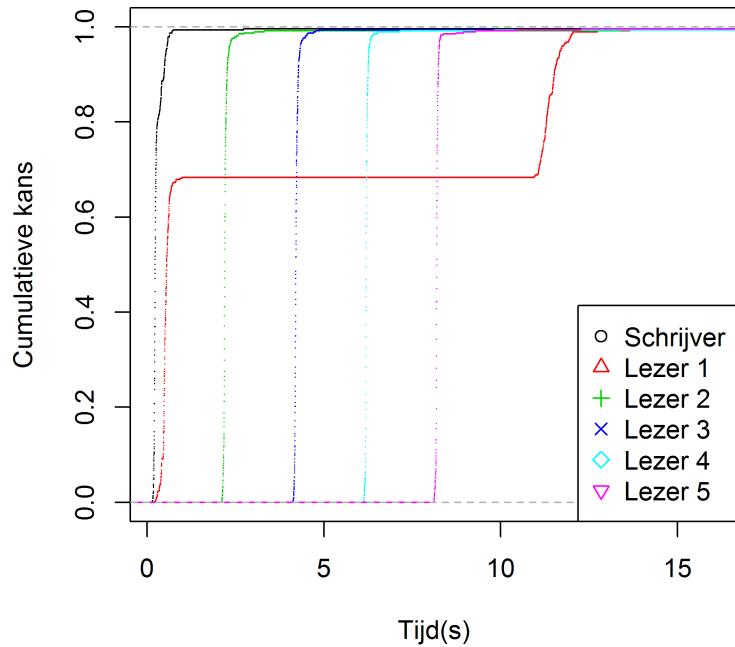
Figuur 4.13: Consistentie van MongoDB: Overzicht van startmoment van consistente leesactie voor het lezen op de nearest



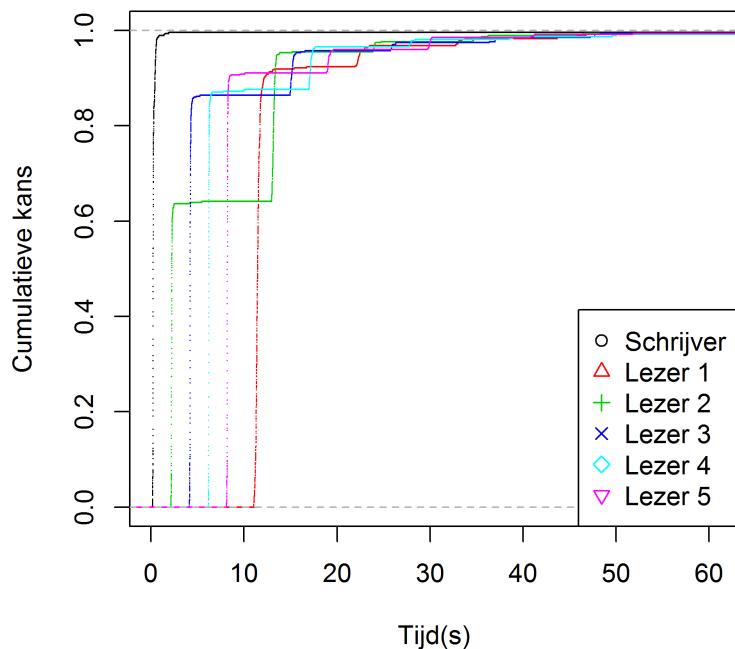
Figuur 4.14: Consistentie van MongoDB: Overzicht van startmoment van consistente leesactie voor het lezen op de primary

#### 4. OBSERVATIES

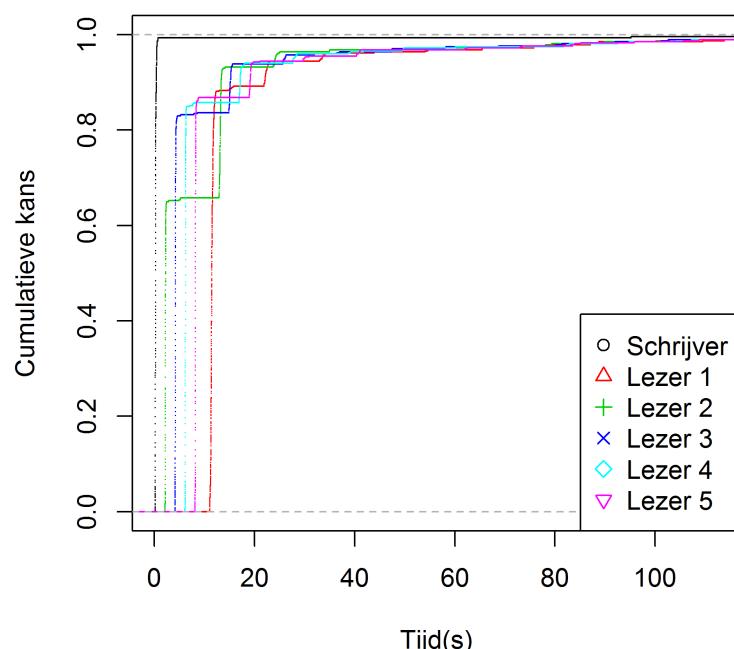
---



Figuur 4.15: Consistentie van MongoDB: Overzicht van startmoment van consistente leesactie voor het lezen op de primarypreferred



Figuur 4.16: Consistentie van MongoDB: Overzicht van startmoment van consistente leesactie voor het lezen op de secondary



Figuur 4.17: Consistentie van MongoDB: Overzicht van startmoment van consistente leesactie voor het lezen op de secondarypreferred.

#### 4.4 Conclusie

In dit hoofdstuk zijn de resultaten van de verschillende testen getoond voor MongoDB, HBase en Pgpool-II. Het gemeten gedrag is verschillend tussen de 3 systemen naar beschikbaarheid toe. Ook in consistentie zijn er verschillen tussen HBase en MongoDB. Opvallend is dat er geen verschil in leesactie is bij de verschillende schrijfconfiguratie als de startmomenten vergeleken worden.

In het volgende hoofdstuk zal deze data verder geanalyseerd worden en verklaringen of hypotheses zullen gesteld worden.

# **Hoofdstuk 5**

## **Analyse van de resultaten**

In dit hoofdstuk worden de observaties in meer detail besproken. Voor elke observatie zal er geprobeerd worden een verklaren te vinden of een hypothese voor te stellen.

Eerst zal de calibratie aan bod komen, vervolgens de beschikbaarheids- en tenslotte de consistentietesten. Bij elke test zullen eerst de verschillende systemen besproken worden om vervolgens een vergelijking te maken.

### **5.1 Calibratie**

Over de calibratie testen valt in het algemeen niet veel af te leiden, de verschillende systemen hebben niet hetzelfde aantal instanties, zo heeft Pgpool-II slechts 3 instanties t.o.v. 6 voor MongoDB.

Enkel op de stijgende variatie van de vertagingen in MongoDB zal dieper ingegaan worden. De reden hiervoor is een lezers/schrijving locking systeem op een gehele database[24]. Hierdoor zorgt een leesactie voor de blokkering van alle schrijfactiviteiten op de database en vice versa. Naar mate er meer gebruikers zijn, kunnen er meer opeenvolgende schrijfoperaties zijn, dit zal de leesacties langer blokkeren. Maar indien alle gebruikers samen lezen, kan dit parallel gebeuren, een grotere variatie in de vertraging treedt hierdoor op.

### **5.2 Beschikbaarheidstest**

Bij de beschikbaarheidstesten blijkt er uit de resultaten dat de verschillende systemen een andere aanpak hebben genomen. Voor elk systeem zullen de 3 testen in detail besproken worden, daarna zal een vergelijking tussen de 3 systemen besproken worden. Belangrijk is dat er hier bewerking aan de basisbelasting uitgevoerd worden, waardoor er data van de verschillende datadistributies gelezen zal worden.

## 5. ANALYSE VAN DE RESULTATEN

---

**HBase** Bij HBase heeft een bepaalde RegionServer de verantwoordelijkheid over een Regio voor een bepaalde tijd. Dit is een sessie dit door HMaster uitgedeeld wordt en bijgehouden wordt in Zookeeper. Deze sessie kan vroegtijdig beëindigd worden of er moet gewacht worden tot deze verlopen is, enkel op die momenten kan er een nieuwe RegionServer aangeduid worden. Dit zorgt voor een duidelijk verschil tussen een zachte stop, een harde stop of een netwerk onderbreking.

De duur van een sessie kan geconfigureerd worden in Zookeeper en staat standaard op 180 seconden. [38].

**Zachte stop** Bij een zachte stop, is er slechts af en toe sprake dat dit merkbaar is, de verklaring hiervoor is dat dit enkel wordt opgemerkt als de RegionServer die op dat moment verantwoordelijke is voor de Region wordt stopgezet. In de testen is dit niet zichtbaar omdat er verschillende opeenvolgende queries worden uitgevoerd, hierdoor zal data van verschillende regio's gelezen worden tot de blokkerende regio wordt gevonden.

Indien een RegionServer wordt stopgezet, nemen de queries tijdelijk meer tijd in beslag. Na het stopzetten van de RegionServer is er in bepaalde gevallen een verhoogde vertraging in beide leesoperaties (scan en lees).

Zodra er een herverdeling is van de Regions over de aanwezige Regionservers, verdwijnt deze verhoogde vertraging.

**Netwerk onderbreking** Bij een netwerk onderbreking, worden de queries tijdelijk stopgezet en falen de queries in tussentijd. Deze onderbreking duurt significant langer dan in het geval van een zachte stop. Dit komt doordat de regio's pas kunnen toegewezen worden na het verlopen van hun sessie.

**Harde stop** Bij het stopzetten van een instantie op de harde manier, zijn er twee reacties: de eerste is vergelijkbaar met deze van een netwerk onderbreking. De andere laat pas opnieuw queries toe na het herstellen van het netwerk verkeer. In een manuele test bleek dit opgelost te zijn na het verbreken van de connectie en het opnieuw verbinden, maar de oorzaak waarom dit slecht af en toe voorkomt, is niet gevonden.

**Herstel van de instantie** Het herstel van de server zal automatisch op een asynchrone manier gebeuren. Er valt te configureren hoeveel data er maximaal per seconde zal worden gesynchroniseerd. Het herstel is niet merkbaar voor de gebruiker bij de testen.

**MongoDB** Bij MongoDB is er tussen de leden van een Replicaset een heartbeat protocol. Indien er gedurende 10 seconden geen antwoord op een heartbeat komt, wordt een server als offline bestempeld. De server die het laatst een bericht van de primary heeft gekregen zal zijn rol overnemen indien deze uitvalt [25]. Dit heeft zijn invloed op de verschillende manieren om een server stop te stopzetten.

**Zachte stop en harde stop** Bij een zachte of harde stop is er een kans van 1 op 3 dat het uitvallen van een instantie zichtbaar is, dit is te verklaren doordat enkel het uitschakelen van de primary een invloed zal hebben op de vertraging. In de standaard modus wordt er enkel gelezen naar en geschreven van de primary. Na de verkiezing van een nieuw primary is er geen verschil in vertraging per soort query ten opzichte van ervoor. Er zijn 2 hypothesen waarom een harde stop op dezelfde manier reageert: bij een harde stop wordt de sessie nog altijd vrijgegeven, of er zijn verkiezingen voor een nieuwe primary voordat de sessie van de oude primary is afgelopen. Deze hypothesen zijn niet getest of verder onderzocht.

**Netwerk onderbreking** Bij een netwerk onderbreking zou het te verwachten zijn dat na maximaal 10 seconde de primary zou veranderen. Onder de aangelegde belasting blijkt dat de database heel de tijd onbeschikbaar tot de primary opnieuw bereikbaar is via het netwerk. Bij het manueel testen blijkt dat er dezelfde foutmelding gegeven wordt als bij het stoppen van de database, maar dat de data onbeschikbaar is. Het volstaat om de verbindingen af te sluiten en opnieuw aan te maken om het probleem op te lossen. Er is geen reden gevonden voor dit gedrag.

**Herstel van de instantie** Het herstel van de server zal automatisch op een asynchrone manier gebeuren. Dit is niet merkbaar voor de gebruikers en de server zal als secondary ingezet worden.

**Pgpool-II** Bij Pgpool-II wordt de status van de PostgreSQL instanties getest wanneer er een gebruiker actief is. Bij het uitvallen van een instantie en opnieuw opstarten terwijl er geen gebruiker verbonden is met Pgpool-II, zal dit niet opgemerkt worden. Daarnaast zijn er verschillende reacties op de geteste scenario's.

Een vereiste bij het herstellen van een instantie is dat er op dat moment geen enkele verbinding met de router instantie is.

**Zachte stop** Bij een zachte stop van een data instantie worden alle verbindingen van de gebruikers met Pgpool-II onmiddellijk verbroken. Nadien kan er terug verbonden worden met Pgpool-II. In deze omgeving gaan nadien de verschillende schrijfoperaties sneller omdat deze niet meer gerepliceerd moeten worden. Een hypothese is dat bij een grote hoeveelheid data instanties dit effect kleiner zal worden.

**Harde stop** Een harde stop reageert hetzelfde als een zachte stop. Dit omdat ook hier de connecties onmiddellijk verbroken zijn. Het besturingssysteem van de data instantie zal antwoorden dat er geen service op de poort aan het luisteren is en Pgpool-II detecteert de fout.

**Netwerk onderbreking** Bij een netwerk onderbreking is er een ander gedrag, de queries wachten op een antwoord maar krijgen dit niet. Hierdoor wordt er gewacht

## 5. ANALYSE VAN DE RESULTATEN

---

op de time-out die standaard 30 seconde is. Na deze tijd worden connecties verbroken. De gebruiker kan opnieuw verbinden en queries uitvoeren.

**Vermindering van schrijfvertraging** De reden tot de vermindering van de schrijfvertraging is te vinden in de manier dat Pgpool-II de replicatie van de queries doet. Deze zullen eerst op de master uitgevoerd worden en vervolgens op de slaves. Bij het wegvallen van een instantie is er nog maar een enkele data server over, hierdoor duurt een schrijfactie maar half zo lang. De leesacties duren ongeveer even lang aangezien dezelfde acties nog steeds genomen worden.

**Herstel van de instantie** Bij het opnieuw inschakelen van een instantie dient in Pgpool-II het herstel handmatig in gang gezet te worden. De data zal van de master naar de instantie gesynchroniseerd worden. In het geval van een grote achterstand zal dit merkbaar zijn omdat het proces aan maximale snelheid wordt uitgevoerd; een grote belasting op de CPU, harde schijf en het netwerk kunnen dus voorkomen voor het lezen, comprimeren en versturen van de data. Om het herstel te voltooien moeten alle connecties naar Pgpool-II op een gegeven moment gesloten worden. In de testen die werden uitgevoerd waren er steeds actief en hierdoor slaagde het herstel niet.

**Conclusie** Hoewel er verschillende reacties zijn tussen HBase en MongoDB, ligt de interne werking vrij dicht bij elkaar, de status van beiden wordt permanent opgevolgd. Bij MongoDB gebeurt dit door de data instanties zelf en kan de parameter niet aangepast worden. Bij HBase is er een extern systeem voor gebruikt waarbij de parameter geconfigureerd worden. Pgpool-II heeft een heel ander systeem door enkel de instanties te controleren op het moment dat er een verbinding is. Daarnaast ondersteunt Pgpool-II ook niet de automatische herstel en komt de handmatige herstel niet tot voltooiing onder constant gebruik, hiervoor zijn beide andere systemen automatischer. Een overzicht van het gedrag bij het stoppen en starten van een instantie , bevinden zich in tabel 5.1 en 5.2.

	Zachte stop	Harde stop	Netwerk stop
HBase	Enkele seconden	Tiental seconden of onbeperkt	Tiental seconden
MongoDB	1/3 van de gevallen, enkele seconden	1/3 van de gevallen, enkele seconden	Enkele seconden tot Onbeperkt
Pgpool-II	Enkele seconden	Enkele seconden	30 seconden

Tabel 5.1: Beschikbaarheid: Overzicht van de reacties bij het stoppen van een instantie

### 5.3 Consistentietest

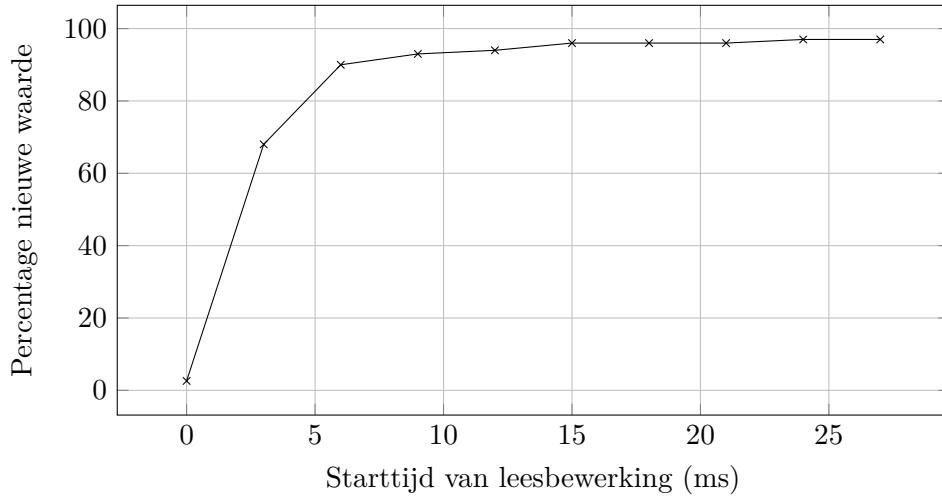
**HBase** HBase garandeert strikte consistentie op een enkel record en hoe deze garantie tot uitvoering wordt gebracht, is zichtbaar in figuur 4.8(b). Een leesbewerking

	Automatisch herstel
HBase	Ja
MongoDB	Ja
Pgpool-II	Nee

Tabel 5.2: Beschikbaarheid: Overzicht van de ondersteuning van automatisch herstel

wordt namelijk op wacht gezet tot de schrijfbewerking voltooid is. Dit valt af te lezen doordat de lijn van het stoppen met schrijven een hogere waarde heeft als het stoppen met lezen en dit zo de hele tijd is. Daarnaast volgt het stoptijdstip van de leesbewerking deze van de schrijfbewerking. Tenslotte kan ook de data in meer detail bekijken worden zoals in figuur 4.9. Hierin is komt naar voor dat het einde van een leesactie het einde van een schrijfactie volgt. Indien een leesactie voltooid is voor de schrijfactie, zal de leesactie een tweede keer moeten lezen voor de correcte data. In figuur 5.2 wordt het lees- en schrijfmodel van HBase uitgelegd aan de hand van de uitleg van Lars Hofhansl[16]. De combinatie van een enkele HRegionServer voor een record en het gebruiken van locks, zorgt ervoor dat atomaire acties op een enkele record succesvol afgedwongen kunnen worden.

Uit de testresultaten blijkt dat indien de leesbewerking te snel verstuurd wordt, er nog geen blokkering van de bewerking zal plaats vinden. Het percentage van de queries dat de nieuwe data zal lezen, bevindt zich in figuur 5.1.



Figuur 5.1: Consistentie: Percentage van de queries dat op een gegeven tijdstip de juiste data leest voor HBase. Het gemiddelde verschil tussen het starten en stoppen van het lezen op een willekeurig record is ongeveer 6ms, in het geval van hetzelfde record kan dit significant langer duren.

**MongoDB** MongoDB biedt strikte consistentie aan als er van de primary gelezen wordt maar er zijn ook andere schrijf- en leesmethodes. Een verschil met HBase is

## 5. ANALYSE VAN DE RESULTATEN

---

### Schrijven

1. Lock de rij(en), om te beschermen tegen concurrente schrijfacties.
2. Haal het huidige schrijfnummer op
3. Voeg aanpassingen toe aan WAL (Write Ahead Log)
4. Pas aanpassing toe op de Memstore (cache geheugen)
5. Commit de transactie, m.a.w. zet het leespunt op het nieuwe schrijfnummer
6. Unlock de rijen

### Lezen

1. Open de lezer
2. Ga naar het huidige leespunt
3. Filter al de Key-Values paren met schrijfnummer > leespunt
4. Sluit de lezer

Figuur 5.2: HBase: Het vereenvoudigde lees- en schrijfmodel voor strikte consistentie in HBase naar Lars Hofhansl[16]

dat het bij alle mogelijke lees- en schrijfmethodes mogelijk is om de nieuwe data al te lezen vooraleer de schrijfbewerking beëindigd is. Een schrijfbewerking wacht op de server nog na het schrijven en vrijgeven van zijn schrijf lock. Een verklaring is hiervoor niet gevonden.

Uit figuren 4.14, 4.15, 4.16, 4.17 en 4.13 kan een analyse gemaakt worden hoeveel kans er is dat een leesbewerking de nieuwe data al zal lezen. Voor lezer 1 tot 5, dit zijn tijdstippen 0, 2, 4, 6 en 8 ms, is een kans berekend ten opzichte van het starttijdstip. Een figuur is getekend voor 4 leesconfiguraties in plot 5.3.

Uit figuren 4.11 blijkt dat er geen significant verschil is tussen de leesacties onder verschillende schrijfgaranties, indien men de starttijdstippen vergelijkt. De schrijfconfiguraties geven geen garanties tijdens het uitvoeren maar enkel na de voltooiing van de schrijfoperatie.

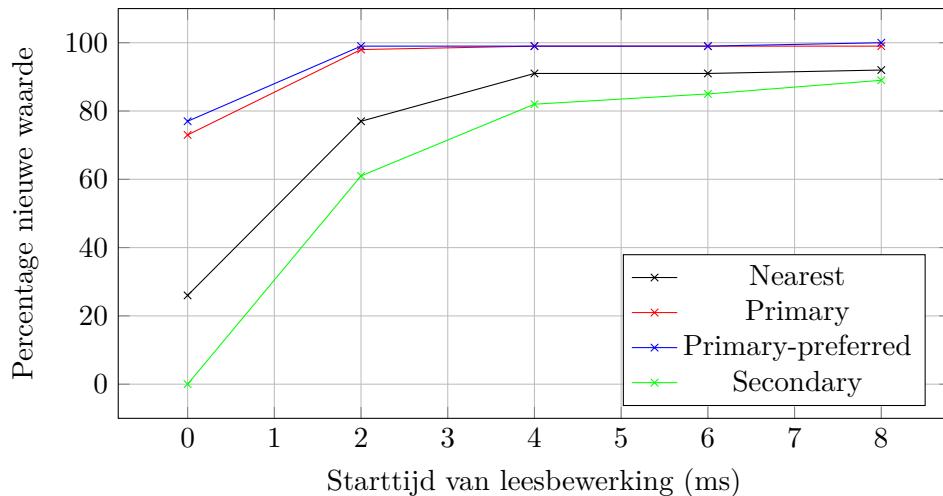
Uit tabel 5.3 blijkt dat het in MongoDB niet is gegarandeerd dat als een lezer de nieuwe waarde leest, dat al de overige lezers dat ook zullen doen. In dit geval was het schrijven nog niet voltooid en een bepaalde lezer leest de nieuwe data al. Maar een bewerking die later gestart is, leest de oude waarde nog. Dit kan verklaard worden doordat het verschillende servers zijn waarop gelezen wordt. In dit geval waren het verschillende lezers en dus verschillende gebruikers, maar de MongoDB driver controleert periodiek welke server het dichtste bij is. Dit controle kan tussen deze 2 bewerkingen gebeuren en men kan van een andere instantie lezen (in het geval de leesconfiguratie niet op primary staat). Er is bij MongoDB géén garantie op monotone leesbewerkingen.

Lezer	Start lezen (ms)	Stop lezen (ms)	Gelezen waarde	Correct?
1	2,200	3,213	125533813315	Nee
	13,426	14,279	125534813315	Ja
3	17,458	18,834	125533813315	Nee
	29,063	29,897	125533813315	Ja

Tabel 5.3: Consistentie: Ruwe data van MongoDB test waarbij inconsistente data wordt gelezen na het lezen van consistente data op verschillende lezers met het lezen via nearest en schrijven via fsync\_safe

Aangezien er in de testomgeving een uniforme netwerkvertraging is naar alle instanties, volgt de data de veronderstelling dat de dichtstbijzijnde node in iets minder 1/3 van de gevallen een primary is en iets meer dan 2/3 een secondary. Met 5% afwijking is het moeilijk te stellen dat deze significant is.

Tenslotte hebben primary en primary-preferred in deze testen geen significante verschillen. Dit komt omdat de primary heel de tijd beschikbaar is.



Figuur 5.3: Consistentie: Percentage van de queries dat van de eerste keer juist de data leest bij 0ms, 2ms, 4ms, 6ms en 8ms voor MongoDB. De gemiddelde vertraging op een onafhankelijke leesoperatie is 1ms.

**Conclusie** Beide database systemen bieden strikte consistentie aan maar hebben een verschillende uitwerking hiervan: bij HBase worden de leesoperaties uitgesteld tot de volledige voltooiing van de schrijfoperatie, bij MongoDB zal de data al vroeger beschikbaar zijn. Beide systemen zijn *session* consistent, *read-your-own-write* en monotoon consistent, indien er bij MongoDB op een primary wordt gelezen.

*Session*, *read-your-own-write*, *casual* en *monotonic* consistentie zijn niet gegarandeerd in MongoDB indien er niet gelezen wordt op een primaire. De MongoDB driver kan

## 5. ANALYSE VAN DE RESULTATEN

---

op ieder moment een andere server kiezen in deze gevallen en kan dus nog oude data lezen.

Een hypothese is dat bij MongoDB het falen van de primary de consistentie garanties zal beïnvloeden, een nieuwe primary kan verkozen worden. Wanneer geen enkele secondary deze update al heeft ontvangen, zou deze dus verloren gaan. Maar een gebruiker zou de data al van de oude primary gelezen kunnen hebben, in dit geval faalt hier de stikte consistentie. Dit gedrag is wel niet getest en bevestigd. HBase heeft deze situatie niet door de keuze om de leesbewerking te verlengen, een gebruiker dient dus langer te wachten op zijn data.

### 5.4 Conclusie

De drie systemen hebben verschillende aanpak naar beschikbaarheid en consistentie. Pgpool-II is het minst geavanceerd systeem door geen automatisch herstel te ondersteunen, maar door de centrale aanpak van de routernode heeft dit systeem geen netwerk verkeer als het niet wordt gebruikt.

MongoDB is een systeem dat weinig configurerbaar is naar het gedrag bij het falen van een instantie, daarentegen zijn er een verschillende configuratiemogelijkheid naar het lees- en schrijfgedrag. Enkel als er gelezen wordt van de primary, zal er strikte consistentie zijn. Het is nog onduidelijk welke garanties er zijn bij het falen van een primary. In normale situaties is het mogelijk om de nieuwe data snel te lezen.

HBase is met behulp van Zookeeper configurerbaar naar het gedrag bij falen van een enkele instantie. De onbeschikbaarheidsperiode kan verkleind of vergroot worden. De consistentiegaranties van HBase zijn strikt voor een enkel record maar dit komt wel voor een prijs: een leesactie wordt uitgesteld indien er een de schrijffactie op dat record uitgevoerd wordt.

# Hoofdstuk 6

## Conclusie

Er zijn veel verschillende database management systemen zijn, waar er veel wordt gedacht aan hun verschil in data en query methodes, zijn er ook verschillen bij hun gedrag in een gedistribueerde omgeving.

Up te daten  
naar latste feed-  
back

In deze thesis is een nieuwe testmethode beschreven en vervolgens uitgewerkt om de consistentie en beschikbaarheidsverschillen van verschillende systemen op een analytische manier te testen. Er is getest naar de beschikbaarheid van de systemen bij het verwachte en onverwacht stopzetten van instantie of netwerk onderbreking. Bij consistentie is er gekeken hoe lang het duurt voor de data beschikbaar is voor de verschillende gebruikers en welke garanties er aangeleverd kunnen worden.

Beide testmethodes zijn uitgewerkt voor HBase en MongoDB, een respectievelijk column en document database management systeem. Voor Pgpool-II, een gedistribueerde uitbreiding van PostgreSQL, zijn enkel de beschikbaarheidstesten uitgevoerd.

Uit de testresultaten blijkt dat hoewel op papier de consistentie tussen HBase en MongoDB gelijk is, zijn er in de praktijk verschillende resultaten. HBase stelt de data beschikbaar voor alle leesgebruikers na de voltooiing van de schrijfbewerking, in tussentijd zullen de leesbewerkingen voor dat record vertraagd worden. MongoDB heeft verschillende configuratie mogelijkheden voor lezen en schrijven, waarbij enkel strikte consistentie is bij het lezen op de primary. MongoDB kiest ervoor om zo snel een query te voltooien met indien mogelijk de nieuwe waarde, ook als de schrijfactie nog niet voltooid is.

Bij de beschikbaarheidstesten is er groot verschil tussen de werking van Pgpool-II en de andere 2 systemen. Pgpool-II zal de status het systeem controleren door tussen de router en de verschillende data instanties een data verbinding op te zetten wanneer een gebruiker verbonden is met het systeem. Het verbreken van deze achterliggende verbinding zal het onderbreken van de gebruikersverbinding als gevolg hebben. Onmiddellijk daarna is het systeem terug beschikbaar.

## 6. CONCLUSIE

---

HBase werkt met sessies van configurerbare duur, tijdens een sessie is een bepaalde server verantwoordelijk voor een deel van de data. Bij een verwachte stop kan deze sessie stopgezet worden en zal de data kort onbereikbaar zijn, bij een onverwachte stop of netwerk onderbreking wordt er gewacht tot na het verlopen van de sessie. Bij een harde stop kan er af en toe voorkomen dat er geen data gelezen wordt als men als gebruiker niet expliciet nieuwe connecties laat aanmaken, bij de andere mogelijkheden gebeurt dit automatisch.

MongoDB werkt met een heartbeat protocol om de status van andere servers te controleren. Bij een verachte of onverwachte stop, is de data kort onbeschikbaar doordat een nieuwe verantwoordelijke moet worden aangeduid. Bij een netwerk onderbreking is, in tegenstelling tot het stoppen van een query, een nieuwe connectie met MongoDB nodig, anders zal de oude data niet gelezen worden.

### 6.1 Verder werk

In deze thesistekst zijn de eerste resultaten en conclusies naar beschikbaarheid en consistentie getrokken. Maar deze test methodes kunnen op meer systemen uitgevoerd worden tot een nieuwe vergelijkingsmethode voor vele database systemen.

Daarnaast kunnen de gebruikte testparameters ook aangepast worden om bepaalde assumpties te verifiëren of mathematische verbanden te zoeken. In de uitgevoerde testen hadden al de verschillende servers met een ping tijd rond de 0.5ms, maar wat is bijvoorbeeld de invloed van deze parameter in de testen, hetzelfde geldt voor het aantal instanties van het DBMS en de belasting op de systemen (verkleint of vergroot het inconsistentie interval bij een hogere belasting?).

Daarnaast kunnen ook de testmethode aangepast worden zoals bij de consistentie test de lezer en schrijver fysiek scheiden. De beschikbaarheidstesten kunnen ook getest worden met verschillende fysieke gebruikers en te onderzoeken of deze hetzelfde gedrag meten.

Als laatste mogelijke uitbreiding, kunnen beide testen gecombineerd worden: verdwijnt er data als een instantie crasht en dit zowel vanuit het perspectief van de schrijver als de lezen. In MongoDB zou het mogelijk kunnen zijn dat een schrijfbewerking nog niet gerepliceerd was naar een secondary maar al wel gelezen was op de primary. Komt dit voor of zijn er mechanismen die dit voorkomen?

# Bijlagen



## Bijlage A

# Bespreking van verschillende DBMS's

In dit hoofdstuk worden verschillende DBMS's in meer detail besproken. RDMBS's en systemen van van NoSQL met uitzondering van graph databases komen aanbod. De besproken systemen zijn:

- Column NoSQL DBMS's: Cassandra, HBase
- Document NoSQL DBMS's: Apache CoucheDB, MongoDB
- Key-Value NoSQL DBMS's: LightCloud (Tokyo), MemCache, Redis, Riak, Project Voldemort
- Relationale DBMS's: MySQL, Pgpool-II (PostgreSQL)

Deze keuze van deze systemen is gebaseerd op de paper van Christophe Strauch [37], Elk van de systemen wordt kort hieronder besproken.

### A.1 Column database

#### A.1.1 Cassandra

Website: <http://cassandra.apache.org/>

Cassandra is een database systeem die gebaseerd is op 2 verschillende systemen, Amazon's Dynamo en Google's Bigtable, wat voor een combinatie van een column- en key-value-based database zorgt.

De query taal is beperkt tot 3 operaties: get, insert en delete [21], waar de laatste waarde in geval van een conflict zal opgeslagen worden.

## A. BESPREKING VAN VERSCHILLENDEN DBMS's

---

De database kan gedistribueerd uitgerold worden waar door middel van partitionering en een consistent hashing algoritme de data verspreid wordt over de verschillende instanties. Om beschikbaarheid van de data te hebben bij een failure, wordt deze gerepliceerd over verschillende instanties met verschillende configuratie modellen.

### A.1.2 HBase

Website: <http://hbase.apache.org/>

HBase is een database systeem die gebaseerd is op Google's BigTable en draait boven op HDFS, Hadoop Distributed File System.

De query taal voor HBase bestaat uit 4 elementen, een get, put en delete als standaard operaties en een scan om over verschillende rijen te gaan.

Voor het gedistribueerd draaien van de database, wordt de database ingedeeld in Regions. Vervolgens is een RegionServer verantwoordelijk voor de data van Regions. Daarnaast zijn er nog Zookeeper en Hadoop die respectievelijk verantwoordelijk zijn voor het management van de instanties en de eigenlijke dataopslag.

## A.2 Document database

### A.2.1 Apache CoucheDB

Website: <http://couchdb.apache.org/>

Apache CoucheDB is een document database systeem waar alles wordt voorgesteld met behulp van JSON. Het systeem kan gevraagd worden door middel van Map-Reduce, de map gebeurd door een *view*, een JavaScript-functie die de gegevens zal selecteren. Nadien kan met een reduce view de data geaggregeerd worden.

Bij het gedistribueerd uitrollen zal de data met consistent hashing over verschillende instanties verdeeld worden waar elke instantie dezelfde rol heeft. Nu zal CoucheDB enkel updates van data van instantie veranderen en niet data automatisch verdelen. Ook is het mogelijk om een exacte replica van de ene naar de andere instantie te sturen, dit wordt bijvoorbeeld handig indien documenten naar een laptop gesynchroniseerd worden om later offline te kunnen werken.

In een gedistribueerde omgeving ziet CouchDB conflicten niet als een uitzondering maar als een normale omstandigheid. Wel zullen updates atomisch per rij afgewerkt worden op een enkele instantie, zodat hier geen conflict in kan bestaan. Maar indien een conflict optreedt, is het aan de bovenliggende applicatie om deze af te handelen.

### A.2.2 MongoDB

Website: <http://www.mongodb.org/>

MongoDB is een document database systeem waar de data wordt voorgesteld aan de

hand van BSON, een binaire vorm vergelijkbaar met JSON. Data kan ingegegeven worden via JSON aangezien er een eenvoudige map mogelijk is.

Er is een uitgebreide query taal, waar er naast het invoegen, verwijderen en opvragen van een document ook talrijke zoekparameters meegegeven kunnen worden: dit gaat van zoeken op een enkel veld tot conjuncties, sorteren, projecties, ...

MongoDB kan in een gedistribueerde omgeving opgezet worden met een opsplitsing tussen het redundant opslaan van data en het verdelen van data. Het redundant opslaan wordt toepast door het combineren van instanties in een ReplicaSet waar er een master-slave configuratie is. Daarnaast kan data ook verdeeld worden over verschillende instanties of replica sets, dit kan door middel van het configureren van shards. Conflicts worden opgevangen door de master waar er telkens een meerderheid van de instanties nodig is om deze te verkiezen.

## A.3 Key-Value database

### A.3.1 LightCloud (Tokyo)

Website: <http://opensource.plurk.com/LightCloud/>

LightCloud is een gedistribueerde uitbreiding van Tokyo Tyrant. Tokyo Tyrant is op zijn beurt een uitbreiding op Tokyo Cabinet en voegt de mogelijkheid tot externe connecties aan Cabinet toe. Cabinet is het basis pakket.

De query taal is gelimiteerd tot 5 operaties: get, put, delete, add en een iterator om over de keys te gaan. Met add wordt er data aan een bestaand element toegevoegd.

LightCloud levert een gedistribueerde database met master-master synchronisatie. Met behulp van een consistent hashing algoritme en 2 hash rings, wordt de data verdeeld over verschillende instanties met de nodige redundantie. De eerste ring is verantwoordelijk voor de lookups oftewel het lokaliseren van de keys, de storage ring is verantwoordelijk voor het opslaan van de verschillende waarden.

### A.3.2 MemCacheDB

Website: <http://memcachedb.org/>

MemCacheDB is een database systeem dat gebaseerd is op MemCache met de aanpassing dat het geen caching systeem meer is maar een systeem met permanente opslag gebouw op de berkeley database. Het data model is eenvoudig en heeft voor elke key een enkele waarde, verschillende kolommen worden niet ondersteund bij een enkele waarde.

De query mogelijkheden zijn beperkt tot get, put en delete van een waarde. In het geval een key meerdere keren geschreven wordt, zal de laatste waarde teruggegeven worden.

## A. BESPREKING VAN VERSCHILLENDEN DBMS's

---

### A.3.3 Redis

*Website: <http://www.redis.io/>*

Redis is een key-value database met de mogelijkheid tot opslaan van complexe datastructuren zoals lijsten, sets en mappen. Naast de standaard instructies om een enkele waarde toe te voegen, zijn er specifieke commando's om operaties op de complexere objecten te doen. Redis biedt ook ondersteuning voor transacties en heeft deze de mogelijkheid tot expire, hierdoor zal een waarde automatisch vergeten worden na een meegegeven tijd.

De database wordt volledig in geheugen geplaatst maar ondersteunt 2 soorten van persistentie, oftewel door middel van RDB, oftewel met een AOF log. Bij RDB worden er over tijd snapshots gemaakt van de database en weggeschreven op harde schijf. In het geval van AOF wordt elke schrijfoperaties weggeschreven en kan de database opgebouwd worden met behulp van deze lijst.

Tenslotte heeft Redis momenteel een relatief beperkte mogelijkheid tot een gedistribueerde database. Het is mogelijk om data over verschillende instanties te distribueren met behulp van sharding welke op voorhand gedefinieerd dient te worden en is er ook de mogelijkheid tot master-slave opstelling met automatische failure detection. De laatste is nog wel in beta, al is het mogelijk om deze te gebruiken. Tenslotte is er in de toekomst meer ondersteuning op komst met behulp van Redis Cluster waar data automatisch verspreid wordt over verschillende instanties.

### A.3.4 Riak

*Website: <http://basho.com/riak/>*

Riak is een key-value database met de mogelijkheid tot opslaan van strings, JSON en XML. Daarnaast heeft deze standaard operaties maar hier enkele uitbreidingen op gemaakt. Allereerst is het mogelijk om secundaire indexen te definiëren op de elementen, MapReduce toe te passen en een full-text search.

Riak is gebouwd om gedistribueerd te draaien waar al de instanties evenwaardig zijn. Data wordt verdeeld over de verschillende instanties en elk element wordt standaard op 3 verschillende instanties opgeslagen. Indien een bepaalde instantie faalt, wordt dit met een gossiping algoritme verspreid over de verschillende instanties waarmee een naburige instantie overneemt. Daarnaast is er automatische recovery indien een instantie terug online komt.

### A.3.5 Project Voldemort

*Website: <http://www.project-voldemort.com/>*

Project Voldemort is een key-value store met enkel 3 basis operaties: get, put en delete met de mogelijkheid voor als keys en values strings, serializable objecten, protocol buffers of raw byte arrays te gebruiken.

Deze database ondersteunt verschillende modes van distributie. De opbouw bestaat uit verschillende lagen, elk met hun eigen gedefinieerde functie. Met behulp van deze lagen kan de ontwikkelaar extra functionaliteit toevoegen met behulp van een extra laag om de applicatie meer te finetunen naar zijn uitwerking. Data wordt verdeeld met behulp van consistent hashing over de verschillende servers, waarbij data verschillende keren wordt bijgehouden om ervoor te zorgen dat de data nog beschikbaar is in het geval van falen.

## A.4 Relationale database

### A.4.1 MySQL

*Website: <http://www.mysql.com/>*

MySQL is een relationele database waarin data kan voorgesteld worden in verschillende vormen, beginnend met een bool tot een blok tekst. Daarnaast zijn de query mogelijkheden uitgebreid.

De uitbreiding van een gedistribueerd systeem is bij MySQL ingebouwd door middel van een Master-Slave configuratie. Als mysqlfailover een faal detecteert in één van de slaven, zal de database verder werken, bij het falen van de master zal een nieuwe master handmatig aangeduid moeten worden. Ook de recovery moet handmatig gestart worden, waarna indien gewenst de originele master opnieuw als master kan gezet worden (bv. omdat deze de krachtigste computer is).

### A.4.2 Pgpool-II (PostgreSQL)

*Website: <http://www.pgpool.net/>*

PostgreSQL is een relationele database en heeft soortgelijke specificaties als MySQL op een enkele computer, verschillende soorten data kunnen voorgesteld worden met uitgebreide query mogelijkheden.

Enkel als de database ook gedistribueerd moet uitgerold worden, is er een verschil. Bij PostgreSQL is er standaard geen ondersteuning hiervoor maar moet er op externe elementen vertrouwd worden. Er bestaan verschillende componenten soorten systemen, maar het meeste uitgebreide pakket is Pgpool-II. Deze ondersteund load-balancing, een vergelijking van de systemen kan gevonden worden op de wiki van PostgreSQL [33].

Pgpool-II heeft verschillende mode, zoals parallel mode waar de data verdeeld wordt over verschillende instanties of replicatie waar de data op meerdere instanties wordt opgeslagen zodat deze nog beschikbaar is bij het falen van een enkele instantie.



## Bijlage B

# Overzicht van gedetailleerde implementatie keuzes

In dit hoofdstuk bevinden zich het deployment van de testomgeving en de verschillende configuratie mogelijkheden voor de testuitbreidingen en configuratie van de testen.

Naam	eenheid
ID	String
Starttijdstip	milliseconden
Commando	String

Tabel B.1: Configuratie van event support

Naam	eenheid
ID	String
Starttijdstip	milliseconden
Duur van de actie	microseconden
Gestart?	Boolean
Beëindigd?	Boolean
Exit code	Integer

Tabel B.2: Uitvoer van event support

## B. OVERZICHT VAN GEDETAILLEERDE IMPLEMENTATIE KEUZES

---

<b>Naam</b>	<b>eenheid</b>	<b>Omschrijving</b>
consistencyTest	Boolean	Het activeren van de consistentie test
addSeparateWorkload	Boolean	Het toevoegen van een basis belasting
starttime	Milli-seconden	Het startmoment van de consistentie test
readThreads	Integer	Het aantal lees gebruikers
consistencyDelayMillis	Milli-seconden	Het interval waarin een lees gebruiker opnieuw het record leest
newrequestperiodMillis	Milli-seconden	Het interval waarin een schrijf gebruiker opnieuw een record schrijft
insertProportion-	Float	Het percentage van schrijfacties dat een nieuw record invoegt
ConsistencyCheck	( $0 \leq x \leq 1$ )	Het percentage van schrijfacties dat een record aanpast
updateProportion-	Float	Stop zodra de eerste keer een correct record is gelezen
ConsistencyCheck	( $0 \leq x \leq 1$ )	De maximale afwijking tussen de eigenlijke start van de query en het geplande moment
stopOnFirstConsistency	Boolean	De maximale tijd dat een leesactie geprobeerd wordt
maxDelayConsistency-	Micro-seconden	
BeforeDropInMicros		
timeoutConsistency-	Micro-seconden	
BeforeDropInMicro		

Tabel B.3: Configuratie van de consistentie testen met uitzondering van de locatie voor de logbestanden.

<b>Naam</b>	<b>eenheid</b>	<b>Omschrijving</b>
Tijd	Microseconden	Het moment dat de schrijfactie moest starten
GebruikersID	R/W-Integer	Het id van de gebruiker (W-0, R-0, R-1, ..)
Start	Microseconden	Het moment dat actie is begonnen
Vertraging	Microseconden	De tijd dat de actie heeft geduur
Waarde	String	De gelezen of geschreven waarde

Tabel B.4: Uitvoer van een enkel query in de consistentie testen

---

<b>Stoppen</b>	
<b>Wat</b>	<b>Commando</b>
Zachte stop	service {{service-name}} stop
Harde stop	kill -KILL {{process Id}}
Netwerk onderbreken	iptables -A OUTPUT -d 0.0.0.0/0 -j DROP
<b>Heropstarten</b>	
<b>Wat</b>	<b>Commando</b>
Zachte start	service {{service-name}} restart
Harde start	service {{service-name}} restart
Netwerk herstellen	iptables -D OUTPUT 1
<b>Speciale commando's</b>	
<b>Wat</b>	<b>Commando</b>
Pgpool-II (Online recovery)	/usr/local/bin/pcp_recovery_node -d 10 {{pgpool host}} {{port}} {{gebruikersnaam}} {{wachtwoord}} {{node nummer}}

Tabel B.5: Beschikbaarheidstesten: Overzicht van de commando's voor het stoppen en starten in de verschillende modes.



## Bijlage C

# Figuren van de observaties

Dit hoofdstuk bevat de testdata op een grafische manier voorgesteld. Dit is enkel een selectie van de figuren, al de data en figuren kunnen gevonden worden op <https://github.com/thuys/YCSB-Testdata>.



## Bijlage D

# Opstellen van de testomgeving en uitvoering van de testen

Dit hoofdstuk bestaat uit 2 gedeelten, eerst komt een uitgebreide beschrijving aanbod voor het opzetten van de verschillende systemen met behulp van IMP. Nadien zal er uitgelegd worden hoe de volledige test uitgevoerd wordt.

### D.1 Uitwerking in IMP

In deze sectie zal voor de verschillende systemen de automatisering van installatie en configuratie met behulp van IMP uitgelegd worden. Er zal steeds de afhankelijkheden gegeven worden, een domeinmodel, uitleg bij het domeinmodel en voorbeeld configuratie gegeven worden.

De automatisatie van installatie is ontwikkeld en getest met Fedora 18 en 20, op andere distributies en versies is er niet getest. Elk systeem maakt gebruik van *ip::services::Server*, een instantie hiervan is een (virtuele) machine met een IP adres en besturingssysteem.

Bij elke instantie is het verplicht om de firewall uit te zetten en SELinux op permissive te zetten. Dit kan met behulp van de volgende commando's:

```
systemctl stop firewalld.service
systemctl disable firewalld.service
setenforce 0
sed -i "s/SELINUX=enforcing/SELINUX=permissive/g" /etc/
    sysconfig/selinux
sed -i "s/SELINUX=enforcing/SELINUX=permissive/g" /etc/
    selinux/config |
```

### D.1.1 HBase

Link: <https://github.com/thuys/hbase>

Benodigde IMP modules: std, net, ip, redhat, hosts en yum.

De installatie en configuratie is gebeurd aan de hand van de uitleg en yum-repository van Cloudera<sup>1</sup>.

#### Domein model en uitleg

Het domeinmodel is te zien in figuur D.1.

**HBaseMaster** Dit is de implementatie van de HMaster, dient toegewezen worden aan een host met java installatie. De poort is de poort waarop de HMaster actief is.

**HRegion** Dit is de implementatie van de HRegionServer, dient toegewezen worden aan een host met java installatie. De poort is de poort waarop de HRegionServer actief is.

**HadoopHDFS** Dit is de implementatie van de HDFS namenode, dient toegewezen worden aan een host met java installatie. De poort is de poort waarop de namenode actief is, de directory is de directory voor HBase en de nameDir de lokatie waar de data op harde schijf weggeschreven zal worden.

**HadoopDatanode** Dit is de implementatie van de HDFS datanode, dient toegewezen worden aan een host met java installatie. De poort is de poort waarop de namenode datanode is, de directory is de directory voor HBase en de dataDir de lokatie waar de data op harde schijf weggeschreven zal worden.

**Zookeeper** Dit is de implementatie van een enkele Zookeeper. Bij het toewijzen van meerdere aan een cluster zullen de Zookeepers een cluster vormen.

**Javahost** Dit is een server waar Java is op geïnstalleerd.

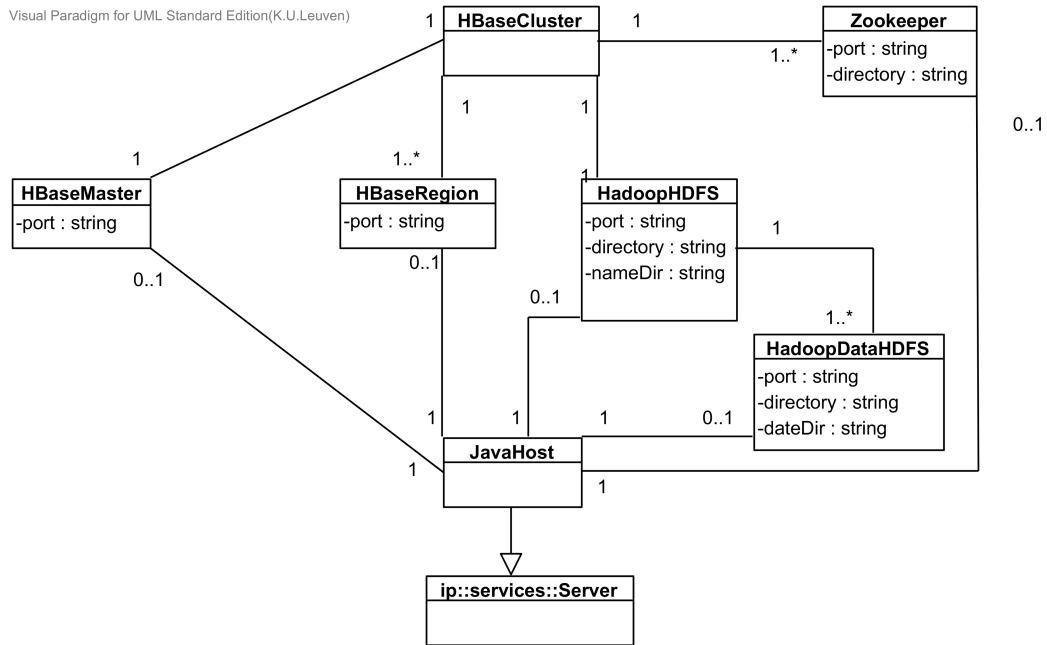
#### Voorbeeld configuratie

De configuratie voor de testomgeving gaat als volgt:

```
vmHB1 = ip :: Host(name = "vmhb1", os = "fedora-18", ip = "172.16.32.9")
vmHB2 = ip :: Host(name = "vmhb2", os = "fedora-18", ip = "172.16.32.10")
```

<sup>1</sup><http://www.cloudera.com/content/cloudera-content/cloudera-docs/CDH4/4.2.0/CDH4-Installation-Guide/CDH4-Installation-Guide.html>

## D.1. Uitwerking in IMP



Figuur D.1: HBase: Domeinmodel HBase in IMP

```

vmHB3 = ip :: Host( name = "vmhb3" , os = "fedora-18" , ip = "
172.16.32.11 ")
vmHB4 = ip :: Host( name = "vmhb4" , os = "fedora-18" , ip = "
172.16.32.12 ")
vmHB5 = ip :: Host( name = "vmhb5" , os = "fedora-18" , ip = "
172.16.32.13 ")

hbaseHost1 = hbase :: HBaseBasic( host = vmHB1)
hbaseHost2 = hbase :: HBaseBasic( host = vmHB2)
hbaseHost3 = hbase :: HBaseBasic( host = vmHB3)
hbaseHost4 = hbase :: HBaseBasic( host = vmHB4)
hbaseHost5 = hbase :: HBaseBasic( host = vmHB5)

master = hbase :: HBaseMaster( host = hbaseHost1)
region1 = hbase :: HBaseRegion( host = hbaseHost2)
region2 = hbase :: HBaseRegion( host = hbaseHost3)
region3 = hbase :: HBaseRegion( host = hbaseHost4)
region4 = hbase :: HBaseRegion( host = hbaseHost5)

dataNode1 = hbase :: HadoopDataHDFS( host=hbaseHost2)
dataNode2 = hbase :: HadoopDataHDFS( host = hbaseHost3)
dataNode3 = hbase :: HadoopDataHDFS( host = hbaseHost4)

```

```
dataNode4 = hbase::HadoopDataHDFS(host = hbaseHost5)
hdfs = hbase::HadoopHDFS(host = hbaseHost1, dataNodes = [
  dataNode1, dataNode2, dataNode3, dataNode4])

zookeeper1 = hbase::Zookeeper(host = hbaseHost1, number = "1")

hbaseCluster = hbase::HBaseCluster(masters = [master], regions =
  =[region1, region2, region3, region4], zookeepers = [
  zookeeper1], hdfs = hdfs)
```

### D.1.2 MongoDB

Link: <https://github.com/thuys/mongodb>

Benodigde IMP modules: std, net, ip, redhat, hosts en yum.

De installatie en configuratie is gebeurd aan de hand van de uitleg en yum-repository van MongoDB<sup>2</sup>.

#### Domein model en uitleg

Het domeinmodel is te zien in figuur D.2.

**MongoDB** is een server in het IMP model en is verantwoordelijk voor het installeren van de basis van MongoDB. Hierna zijn basis commando's voor connectie te maken met een MongoDB instantie beschikbaar.

**MonogDBServer** is een server in het IMP model en is verantwoordelijk voor het installeren van de MongoDB server.

**MongoDBNode** is de implementatie van een data instantie, maximaal 1 per server. Indien gelinkt met een replica set zal deze als een deel van een replica set worden geïnitialiseerd, anders als een zelfstandige instantie.

**MongoDBReplicaSet** is de voorstelling van een replica set, dit wordt niet aan een specifieke server toegewezen.

**MongoDBReplicaSetController** is verantwoordelijk om de replica set te initialiseren. Belangrijk is dat indien er een uitbreiding is van de set, de node verbonden met de controller een reeds geïnitialiseerde node is.

---

<sup>2</sup><http://docs.mongodb.org/manual/tutorial/install-mongodb-on-red-hat-centos-or-fedora-linux/>, <http://docs.mongodb.org/manual/tutorial/deploy-replica-set-for-testing/> en <http://docs.mongodb.org/manual/tutorial/deploy-shard-cluster/>

**MongoDBConfigServer** is de implementatie van een configuratie server, 1 of 3 servers zijn nodig per cluster.

**MongoDBAccessServer** is de implementatie van mongos, minstens 1 is nodig maar meer kunnen gebruikt worden.

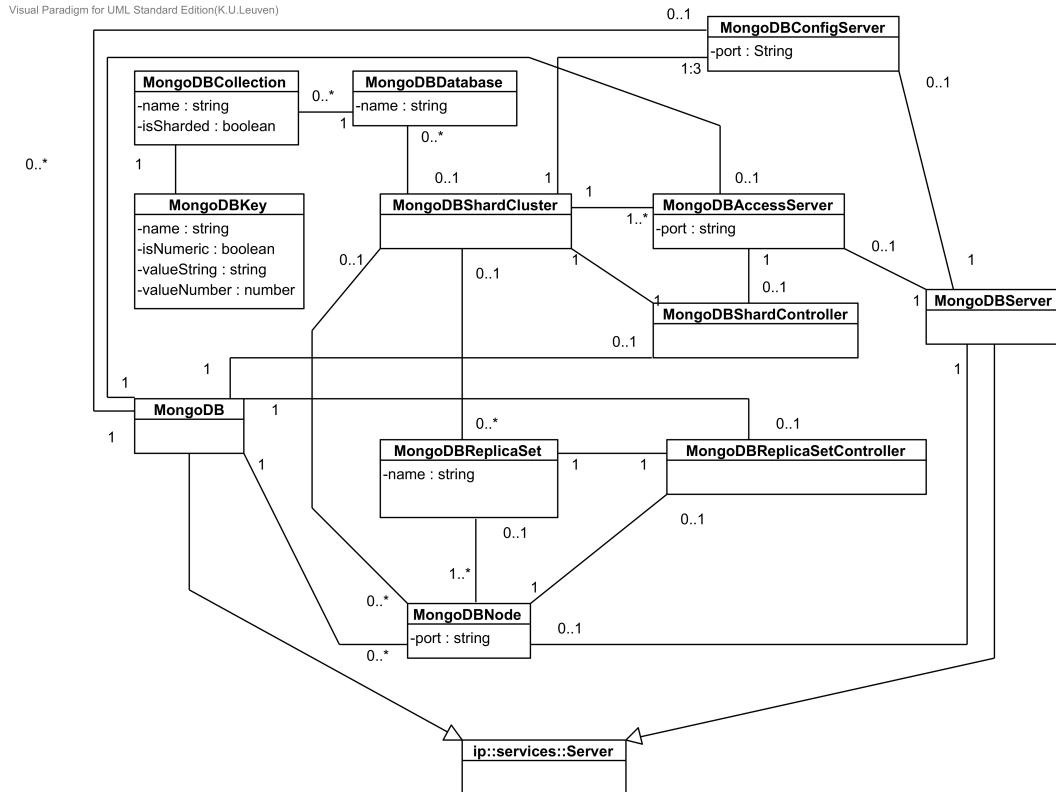
**MongoDBShardCluster** is de voorstelling van een cluster van shards, er kunnen zowel alleenstaand instanties als replica sets aan toegevoegd worden.

**MongoDBShardController** is verantwoordelijk om de cluster te initialiseren met de verschillende shards, databases, collecties en keys.

**MongoDBDatabase** is de voorstelling van een database.

**MongoDBCollection** is de voorstelling van een collectie, indien verbonden met een cluster via een database zal deze gedeeld worden over de verschillende shards.

**MongoDBKey** is de wijze waarmee een collectie verdeeld wordt over de verschillende shards.



Figuur D.2: MongoDB: Domeinmodel MongoDB in IMP

### Voorbeeld configuratie

De configuratie voor de testomgeving gaat als onderstaand. Bij de uitrol van IMP gaat dit verschillende keren uitgevoerd moeten worden omdat eerst de MongoDBNodes moeten draaien, vervolgens kunnen de replicasetts aangemaakt worden, daarna kunnen de replicasetts pas toegevoegd worden in de cluster.

In IMP was het nog niet mogelijk om een te zeggen dat x uitgevoerd moet zijn op een andere instantie, vooraleer y kan uitgevoerd worden, ondertussen is dit mogelijk door de thesis van Harm De Weirdt[10] waar de nieuwe installatie beschikbaar is op <https://github.com/Foezjie/mongodb> maar hierbij dient ook gebruik gemaakt te worden van zijn IMP installatie.

Met het ontbreken hieraan kan het zijn dat er 3 keer een volledige IMP deploy uitgevoerd moet worden.

```
vmMDB1 = ip :: Host( name = "vmmdb1" , os = "fedora -18" , ip = "172.16.32.45" )
vmMDB2 = ip :: Host( name = "vmmdb2" , os = "fedora -18" , ip = "172.16.32.46" )
vmMDB3 = ip :: Host( name = "vmmdb3" , os = "fedora -18" , ip = "172.16.32.47" )
vmMDB4 = ip :: Host( name = "vmmdb4" , os = "fedora -18" , ip = "172.16.32.48" )
vmMDB5 = ip :: Host( name = "vmmdb5" , os = "fedora -18" , ip = "172.16.32.49" )
vmMDB6 = ip :: Host( name = "vmmdb6" , os = "fedora -18" , ip = "172.16.32.50" )

mongo1 = mongodb :: MongoDB( host = vmMDB1)
mongo2 = mongodb :: MongoDB( host = vmMDB2)
mongo3 = mongodb :: MongoDB( host = vmMDB3)
mongo4 = mongodb :: MongoDB( host = vmMDB4)
mongo5 = mongodb :: MongoDB( host = vmMDB5)
mongo6 = mongodb :: MongoDB( host = vmMDB6)

mongo1Server = mongodb :: MongoDBServer( host=vmMDB1)
mongo2Server = mongodb :: MongoDBServer( host=vmMDB2)
mongo3Server = mongodb :: MongoDBServer( host=vmMDB3)
mongo4Server = mongodb :: MongoDBServer( host=vmMDB4)
mongo5Server = mongodb :: MongoDBServer( host=vmMDB5)
mongo6Server = mongodb :: MongoDBServer( host=vmMDB6)

mongoN1 = mongodb :: MongoDBNode( host=mongo1 , server=
    mongo1Server )
mongoN2 = mongodb :: MongoDBNode( host=mongo2 , server=
    mongo2Server )
```

```

mongoN3 = mongodb::MongoDBNode( host=mongo3 , server=
    mongo3Server )
mongoN4 = mongodb::MongoDBNode( host=mongo4 , server=
    mongo4Server )
mongoN5 = mongodb::MongoDBNode( host=mongo5 , server=
    mongo5Server )
mongoN6 = mongodb::MongoDBNode( host=mongo6 , server=
    mongo6Server )

set1 = mongodb::MongoDBReplicaSet( name="rep11" , nodes = [
    mongoN1 , mongoN2 , mongoN3 ] )
set2 = mongodb::MongoDBReplicaSet( name="rep12" , nodes = [
    mongoN4 , mongoN5 , mongoN6 ] )

controller1 = mongodb::MongoDBReplicaSetController( host=
    mongo1 , replicaSet = set1 , connectingNode = mongoN1 )
controller2 = mongodb::MongoDBReplicaSetController( host=
    mongo4 , replicaSet = set2 , connectingNode = mongoN4 )

mongoDBCluster = mongodb::MongoDBShardCluster( replicaSets =
    [ set1 , set2 ] )
shardController = mongodb::MongoDBShardController( host=
    mongo5 , accessServer = access3 , shardCluster =
    mongoDBCluster )

access1 = mongodb::MongoDBAccessServer( host=mongo2 , server=
    mongo2Server , shardCluster = mongoDBCluster )
access2 = mongodb::MongoDBAccessServer( host=mongo3 , server=
    mongo3Server , shardCluster = mongoDBCluster )

config1 = mongodb::MongoDBConfigServer( host=mongo2 , server=
    mongo2Server , shardCluster = mongoDBCluster )

databaseYCSB = mongodb::MongoDBDatabase( name="ycsb" ,
    shardCluster = mongoDBCluster )
collectionYCSB = mongodb::MongoDBCollection( name="usertable" ,
    database = databaseYCSB )
keyYCSB = mongodb::MongoDBKey( name = "_id" , valueString = "
    hashed" , collection = collectionYCSB )

```

### D.1.3 Pgpool-II

Link: <https://github.com/thuys/postgresql>

Benodigde IMP modules: std, net, ip, redhat, hosts en yum.

De installatie en configuratie is gebeurd aan de hand van de uitleg Pgpool-II<sup>3</sup>.

### Domein model en uitleg

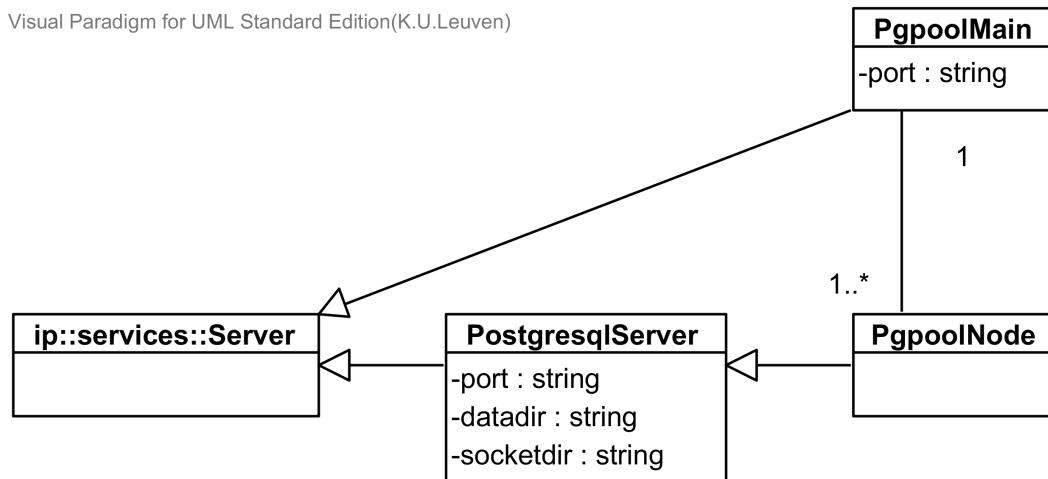
Het domeinmodel is te zien in figuur D.3.

**PgpoolMain** Dit is de implementatie van de Pgpool-II router node.

**PgpoolNode** Dit is de implementatie van de Pgpool-II data node die een uitbreiding is van de standaard PostgreSQL installatie.

**PostgresqlServer** Dit is de implementatie van de standalone PostgreSQL server.

Visual Paradigm for UML Standard Edition(K.U.Leuven)



Figuur D.3: Pgpool-II: Domeinmodel Pgpool-II in IMP

### Voorbeeld configuratie

De configuratie van Pgpool-II gebeurt in verschillende stappen: shell code, IMP uitrol, extra configuratie stap, IMP uitrol.

De eerste shell code bestaat erin om de SELinux volledig uit te schakelen:

```

systemctl stop firewalld.service
systemctl disable firewalld.service
echo "SELINUX=disabled SELINUXTYPE=targeted" > /etc/selinux/
config
  
```

<sup>3</sup><http://pgpool.projects.pgfoundry.org/pgpool-II/doc/tutorial-en.html/>

```
echo "SELINUX=disabled SELINUXTYPE=targeted" > /etc/
sysconfig/selinux
```

De configuratie voor de testomgeving gaat als volgt in IMP:

```
vmPG1 = ip :: Host(name = "vmpg1", os = "fedora-18", ip =
172.16.32.51")
vmPG2 = ip :: Host(name = "vmpg2", os = "fedora-18", ip =
172.16.32.52")
vmPG3 = ip :: Host(name = "vmpg3", os = "fedora-18", ip =
172.16.32.53)

pgNode1 = postgresql::PgpoolNode(host = vmPG1)
pgNode2 = postgresql::PgpoolNode(host = vmPG2)

pgMaster = postgresql::PgpoolMain(host = vmPG3, pgpoolNodes
= [pgNode1, pgNode2])
```

De configuratie bestaat erin om al de verschillende nodes van Pgpool-II, ongeachte of dit routers of datanodes zijn, ssh toegang te geven tot elkaar server via ssh met root en postgres als gebruikers. Deze verbinding al een keer gemaakt zijn want een bericht dat de sleutel nu mee is opgeslagen is voldoende om de online recovery te doen falen.

Hierna kan de IMP uitrol nog een keer gebeuren en het systeem zou moeten werken.

#### D.1.4 YCSB

*Link: <https://github.com/thuys/ycsb>*

Benodigde IMP modules: std, net, ip, redhat, hosts, yum, git, hbase, mongodb, pgpool-II

De installatie en configuratie is gebeurd aan de hand van de uitleg van YCSB<sup>4</sup>.

#### Domein model en uitleg

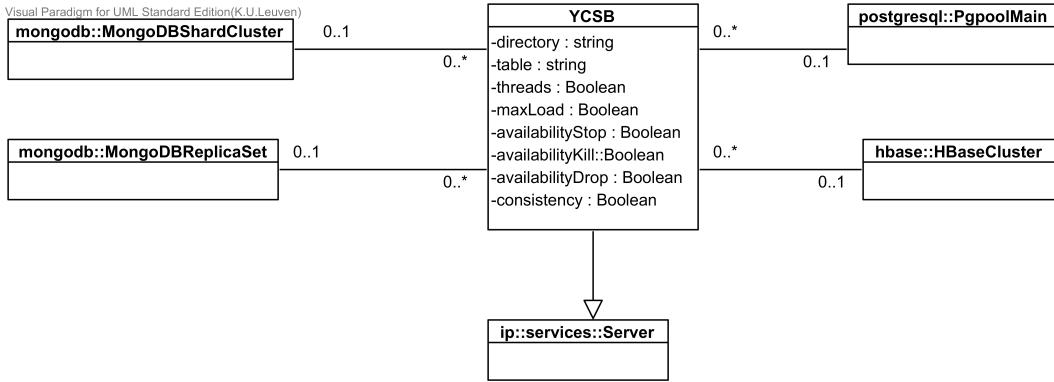
Het domeinmodel is te zien in figuur D.4.

Dit model bevat maar 1 nieuw element en dit is YCSB. Deze dient verbonden te zijn met één van de 3 systemen die hierboven zijn beschreven. Elk van deze systemen zal getest worden voor alle testen die geactiveerd zijn. MongoDB heeft 2 connecties omdat de cluster voor de beschikbaarheidstesten wordt gebruikt en een replicaset

<sup>4</sup><https://github.com/brianfrankcooper/YCSB/wiki/>

## D. OPSTELLEN VAN DE TESTOMGEVING EN UITVOERING VAN DE TESTEN

---



Figuur D.4: YCSB: Domeinmodel YCSB in IMP

voor de consistentie testen. Pgpool-II heeft geen ondersteuning voor de consistentie testen.

### Voorbeeld configuratie

De configuratie voor de testomgeving gaat als volgt in IMP:

```

vmYCSB = ip :: Host( name = "ycsb" , os = "fedora-18" , ip = "
172.16.32.44 ")
ycsb = ycsb :: YCSB( host=vmYCSB , mongoDBCluster ,
mongoDBConsistency = set1 , postgresql = pgMaster , hbase =
hbaseCluster , table="ycsb" ,
threads = false , maxLoad = false ,
availabilityStop = true , availabilityKill = true ,
availabilityDrop = true ,
consistency = true )
  
```

De testen starten door het uitvoeren van directory/scripts/ycsb-script. De resultaten komen in de folder directory/results.

## D.2 Uitvoeren testen

Voor het uitvoeren van de testen dienen de besproken systemen online gebracht te worden, met op het minst YCSB en daarnaast 1 database systeem.

In de installatie folder van YCSB bevindt zich de map *scripts*. Om al de testen uit te voeren dient het bestand *ycsb-script* van die folder uitgevoerd te worden. De configuratie van elke script kan aangepast worden in de map *config* van YCSB.

Als het script voltooid is, kan met behulp van de R-code (<https://github.com/thuys/YCSB-R-Scripts>) de grafieken gecreeerd worden. Dit gebeurt door het aan-

passen en uitvoeren van *Plot-consistentie.R* en *Plot-calibratie-en-beschikbaarheid.R*. Belangrijk is om de *basicDir* van beide scripts correct te zetten.



## **Bijlage E**

## **Paper**

De paper met als titel '*CAP in practice: HBase and MongoDB*' bevindt zich in de volgende pagina's.

# CAP in practice: HBase and MongoDB

Thomas Uyttendaele

**Abstract**—This paper introduced a new, quantified method to test distributed database system in regard to their behaviour to availability in case of a failing node and consistency of data from the users perspective. This method is tested on HBase and MongoDB, two consistent and partition tolerant systems, with the most important outcomes included.

## I. INTRODUCTION

New online services and more online users means more data, data and load a single server can't handle. More and more database systems are distributed, for higher availability in case of an unexpected crash but also for horizontal distribution: the data of a single database is spread over different systems. These new services have different requirements; e.g.. an update doesn't need to be visible for all users immediately, it can take time, this behaviour is called eventual consistency.

Over the past years, many new systems have been build on this wave of changes, categorized as NoSQL. Some applications contain a high volume of data, have the need for consistent data and being able to work in a highly distributed environment, this are the CP systems in the CAP Theorem. Two examples that offer these guarantees are HBase and MongoDB. They greatly differ in supported queries on their system, but what happens if you only use the bare essentials and compare their behaviour in a distributed environment going from expected shut downs of instances towards crashes and network partitions?

In this article, a comparison of both these systems on these 3 behaviours will be made. In chapter II a brief overview of the CAP Theorem is given, chapter III discusses HBase and MongoDB on paper. Chapter IV gives an overview of the used test method and chapter V presents the results. The future work is presented in chapter VI, an overview of related work is given in chapter VII and a conclusion is made in chapter VIII.

## II. THE CAP THEOREM[1][2]

The CAP Theorem was introduced by E. Brewer [1] in 2000 and discusses 3 properties of which each network shared-data system can guarantee at most 2: (definitions based on [2])

- (Strict)Consistency: The system acts like there is only single storage
- High availability: The system is available (for updates)
- Partition tolerance: A split in the network let the different partitions still act as a single system.

Designers used this model to explain their design decisions, others used it to compare different system and sometimes it was misused. As E. Brewer explains 12 year after the launch, the "2 of 3" can be misleading.

One of the reasons is that there exist several types of consistency, partition tolerance and different availability guarantees. The choice for the properties needs to be made several times in the different subsystems and the end solution is not black or white.

At first glance, it also looks like partition tolerance has to be implemented and therefore consistency and availability needs to be given up. In the practice, partition splits are only rare and therefore both consistency and availability can be allowed most of the time.

Each of the 3 choices will be discussed in more detail, how their implementation could work, what the influences are and some examples.

### A. CA: Consistency and availability

When forfeiting partition tolerance, these systems provide all the time consistent data available to all nodes, except when there are one or more nodes unavailable. In that case, write requests will be not allowed.

These systems can be build around the 2 phase commit and have cache invalidation protocols. Examples of this types are the typical relational databases roll out in clusters.

### B. CP: Consistency and partition tolerance

A consistent system with partition tolerance will provide all the time the last data, even in the case of network splits. This comes with the loss of the availability of all nodes all the time.

The system can allow operations only on the majority partition. In case multiple splits are present and no partition has a majority of nodes, the whole system can be unavailable. These systems can be build around a master/slave principle where the operations will be directed to the master, the slaves are present to continue operation when the master fails.

In practice, systems like MongoDB, HBase and Redis select CP.

### C. AP: Availability and partition tolerance

In a highly available systems with partition tolerance, is it possible to read inconsistent data. As read and write operations are still allowed when there are different partitions, it is possible that the database has other content depending on the used node. When the split is dissolved, a need for manual conflict resolution can be needed. In case a record is adapted in both partitions, the user will need to choose the correct version.

Example systems following AP are Cassandra, Riak and Voldemort.

### III. OVERVIEW OF HBASE AND MONGODB

In this article, 2 CP systems will be discussed more in detail regarding their choices to forfeit availability and the influence on their behaviour in practice. The systems are HBase and MongoDB, an architectural overview will be given in this section.

#### A. HBase

HBase[3] is an open-sourced, distributed, versioned database designed after Google's BigTable [4]. HBase relies on Zookeeper for the distributed task coordination and the persistent storage can be done on the local hard disk, Hadoop Distributed File System or Amazon S3. In this article is chosen for Hadoop.

HBase nodes exists out of HMasters and HRegionServers, the coordination of the system is done by one HMaster, the handling of data is done by the HRegionServer. To store the data, multiple Hadoop datanode instance should be deployed for data storage, preferable one on each HRegionServer. The data will be replicated to a configurable amount of other nodes, default modus is 3. The data is stored in a table, which is split in one or more regions. A region is leased to a given HRegionServer for a defined time. During this time, only this server will provide the data of the region to the different users. This way the consistency of data can be guaranteed because there is for each record only a single system responsible. Consistency on a single record is provided by a readers/writer lock on a single record for the according queries, this way there is a guarantee to atomicity on a single record, the full procedure is explained by Lars Hofhansl[5].

To be partition tolerant, the partition with the majority of the Zookeeper servers and a HMaster will appoint regions to available HRegionServers, let's call this the data serving partition. In this approach, it is important to place the Zookeeper and HMaster servers in diverse location as otherwise a partition of only management servers will make the whole system unavailable.

In HBase, a node will be able to answer to requests if the node is present in the data serving partition. Only in rare cases that all data copies of Hadoop are stored in unavailable servers, the data will be unreachable. The nodes not in the data serving partition, will be unable to complete any requests.

When a server goes down, he can release the lease in case there is a graceful shut down (the HBase server is notified) and another server can get the lease immediately. In other cases, a new lease can only be given after the decay of the old lease, if the server comes back online in meantime, he will still be responsible.

#### B. MongoDB

MongoDB[6] is an open-sourced, distributed database designed for document storage, this are data entries where the format of each record can be different. According to their website they provide high performance and high availability, but this is incorrect to the given definition in this article.

MongoDB provides data replication and data distribution, the first is done by grouping different MongoDB servers into a ReplicaSet, the second is done by grouping different of these ReplicaSets.

A ReplicaSet exists out of different MongoDB servers whom work as a master/slave configuration. A master is a primary and a slave is called a secondary. The primary is responsible for the write actions, by default a query will succeed once it has a confirmation that the write has been executed on the primary. The read operation will go by default on the primary as well. Both query methods are configurable to give other guarantees, for a write operation there are multiple *write concerns*, it is possible to wait till it has written on hard disc or a number of secondary servers, however all need a primary. For read operation there are multiple *read preferences*, it is possible to read from a secondary or the closest server.

In the default configuration, MongoDB provides a consistency guarantee.

In a ReplicaSet, there is at least half of the ReplicaSet needed for the primary election. As there is always a primary needed, the system has partition tolerance but no high availability, contrary to the statement on the documentation. However, it is possible to read from a secondary but writing is not possible.

The state of the different members of a ReplicaSet is maintained by a heartbeat system: a server is marked as offline if no beat has been received for 10 seconds. In case the primary goes offline, election will be started to re-elect a new one. In other words, a primary has a lease of 10 seconds. This value of 10 seconds is non-configurable.

De data distributions happens by merging replica sets in a cluster. Furthermore, there is the need for access server (as many as you want) and configuration servers (1 or 3). The availability and partition tolerance of the data is the same as in the ReplicaSets as it handled by the ReplicaSets. In case a access server can't reach a primary, another access server will be needed to write data. If a majority of the configuration servers are not reachable, their will be no reconfiguration of the data over the different servers.

#### C. Differences between databases

Both systems provide consistency and partition tolerance and forfeit high availability, but some differences are in their implementation.

First of all, they differ in their handling of partition tolerance, in HBase there are dedicated management servers (HMaster and Zookeeper) to distribute the responsibilities of regions and if the management servers are in a partition with a minority of the data servers, data will be not available. In MongoDB the management of a ReplicaSet is done internally with as result that the write queries will be available in the partition with the majority of the data servers.

Both decisions have their reasons and possible motivations, in HBase it is possible to write every record, as a new region can be created if the old region is unavailable. In MongoDB

it is possible that in sharding you can read all the data from multiple secondaries but only write given ranges of records.

In availability, there is a small difference: where in MongoDB it is possible to read from secondaries, this is impossible in HBase.

In section V, a detailed analyse will be done to the consistency and the behaviour of the systems in the case of network or server failure.

#### IV. TEST METHOD

To test the behaviour of database systems towards consistency and availability, the Yahoo Cloud Serving Benchmarking (YCSB [7]) has been extended with event support for availability and reader-writers for consistency.

Each of this tests follows the same steps: calibrating the system, records are preloaded, the test is started and in the end all the records are removed again and they are executed for HBase and MongoDB. During the calibration, a workload is chosen so there is a medium load on the different databases.

##### A. Event support

The implementation of event support is integrated so it supports the execution of UNIX commandos at specified moments, the execution time and result code is logged.

Before the test are 300 000 records stored in the database to enable the sharding in all database softwares. In these tests were 3 kinds of tests executed of which each one took 900 second. The first action takes place at 300 seconds, the second at 600 seconds.

- Graceful shut down of a data service and restart of the service
- Hard kill of the data service and a restart of the service
- Blocking of all network traffic from and to a server and the allowance of all network traffic

Each element tests the behaviour of the system under different circumstances, the first two checks what happens in case of a respectively planned and unexpected shut down, the latter check the behaviour in case the network fails.

The tests are executed on each of the data nodes of HBase and MongoDB, caching and buffering on the client side are disabled in both tests.

##### B. Consistency support

To implement consistency support, the YCSB software is extended with an extra workload. A graphical representation of an example workload is shown in figure 1. This workload exists out of 1 writer and 4 readers. Each writer will inserts or updates a value in the database with a user-defined pace. The readers will read the record till they read the last written data element, each reader reads in a period defined by the user. The different readers are scheduled uniformly within the reading period.

All this data is logged and gives possibilities to analyse when a record is visible for different users, compared to the time the queries where started or ended.

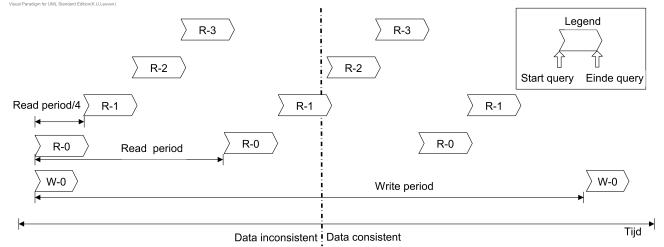


Fig. 1. Example of consistency support for one write period with 1 writer and 4 readers

Before the test, there are 30 000 records stored in the database, each tests takes 500 seconds and results start to be gathered after 30 seconds. In the tests is chosen to write every 0.5s and read with 5 readers every 10ms for MongoDB, for HBase there are 10 readers who read every 30ms.

#### V. RESULTS

To execute the tests, both systems were deployed on a virtual platform of OpenStack. Each instance has 2 CPUs, 4GB RAM and 50GB disc space. The machines are connected with a gigabit Ethernet and an average ping takes 0.4ms ( $\sigma = 0.2$  on 10 000 ping's).

HBase is configured with 5 instances, of which 1 for management (HMaster, Hadoop namenode and Zookeeper) and 4 for data storage (each has a HRegionServer and Hadoop datanode).

MongoDB is configured with 6 instance, grouped by 3 in a ReplicaSet. There was a single configuration server and 3 instances had an access server.

YCSB was deployed on a single instance and used to calibrate the load on both systems. An individual record has 10 fields which each field a size of 100 bytes. The workload existed out of 20% inserts and updates, 40% selects and 20% scans of an uniform spread between 1 and 100. The requests are spread according to Zipfian<sup>1</sup>. The basic load for HBase is an average of 600 queries/second spread over 50 threads, for MongoDB there are 15 threads with a total of 200 queries/second.

##### A. Availability

When reading and writing in the default setting of MongoDB, both MongoDB and HBase will read and write from a leader of a set of data. Both have a lease period for the leader, which can't be set for MongoDB (10 seconds), but can be configured for HBase (default 180 seconds).

In case of a stop of a HBase server in this configuration, the queries will halt till the lease for the region has been expired, this can take between 0 seconds and 180, depending on the moment of the action. In case of a graceful stop of the service, the service will release the lease and a faster handover is possible. During the tests it happened that with a hard stop of the service, the queries are impossible on the connections

<sup>1</sup>Some record are popular, others rare to be used.

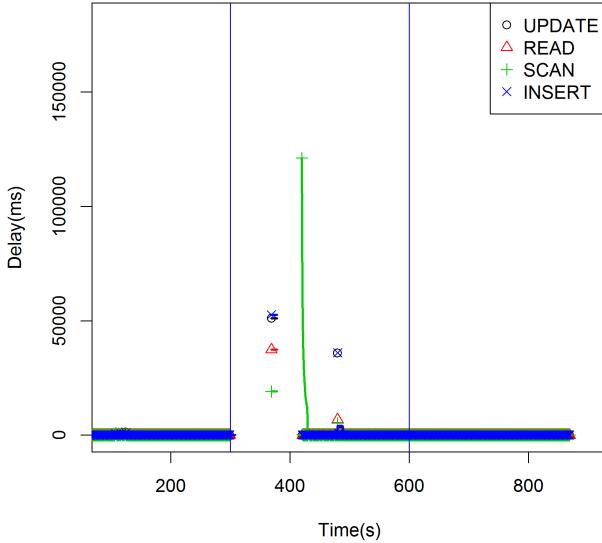


Fig. 2. Example of network partition. The requests block for 110 seconds

till the server is brought back online. In this scenario it was needed to manually dis-en reconnect to the database. A reason why this happens here is not known.

For MongoDB, there is no difference between the graceful or hard stop of an instance; in case it was a secondary, no influences on the latency will be seen. In case a primary was stopped, the data will be in both cases temporarily unavailable for a few seconds. It seems that in a hard stop, there is still a messaging of the shut down to the other nodes. In case of a network interruption, the results are varied, going from no influence, a temporary unavailability for a few seconds, or no completed queries the whole time the server is down. When in the last case, all connections are reinitialized, the queries can happen, when this is not done the table seems empty.

A short overview of all the reactions is shown in table I.

	Graceful stop	Hard stop	Network partition
HBase	Few seconds	Dozens of seconds to unlimited	Dozens of seconds
MongoDB	1/3 of the cases, Few seconds	1/3 of the cases, Few seconds	Few seconds to unlimited

TABLE I

AVAILABILITY: OVERVIEW OF DIFFERENT REACTION WHEN STOPPING AN INSTANCE

### B. Consistency

In consistency HBase and MongoDB have a different approach, in MongoDB it is possible to configure the read and write queries, in HBase there is the choice to use cache at client side. In the tests the cache is disabled but the different options of MongoDB are tested.

HBase: If a record is read while a write query is creating or updating the value, the read query will wait till the completion

Reader	Start time (ms)	Percentage correct read
1	0 ms	2.6%
2	3 ms	68%
3	6 ms	90%
4	9 ms	93%
5	12 ms	94%
6	15 ms	96%
7	18 ms	96%
8	21 ms	96%
9	24 ms	97%
10	27 ms	97%

TABLE II  
CONSISTENCY: PERCENTAGE OF THE QUERIES STARTED AT THE GIVEN TIME MOMENT THAT WILL READ THE NEWEST INSERTED DATA. THE AVERAGE LATENCY IN A RANDOM READ TRANSACTION IS AROUND 6MS.

of the write query and return immediately the correct value. If a read query is send too soon, it will return the old data, but each query returned after the completion of a write query, will return the new data. A graphical representation of this can be seen in figure 3. The stop time of the reader follows the one of the writer, together with extra information, it can be concluded that this is coming from the blocking till the write query has finished.

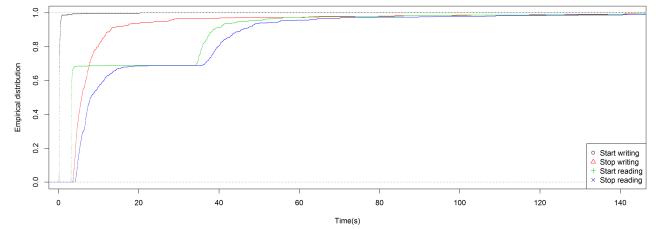


Fig. 3. Consistency: cumulative distribution of queries that read the correct value if reading is started 3ms, 13ms, 23ms. Measurement of over 400 values.

MongoDB: A read query can return the old or new data before the write query has been completed. However a readers/writer lock mechanism is used, it seems that after releasing the lock, the write query still has to execute extra steps. When reading from a secondary or the nearest server, there will be monotone read consistency, except if the driver of MongoDB selects another server between the reads, as there is no guarantee that the other servers already have the value. The process of finding the nearest server happens periodically in the background.

From the test results, the writer configuration has no influence on the speed the replication will have, as the read queries get the same consistency window based on the start moment. The difference is the guarantee a user has when a write query has finished.

An overview of the consistency windows is shown in table II and III for respectively HBase and MongoDB.

### VI. FUTURE WORK

The test run on the different systems are a start to have a quantified approach on the consistency and availability of the

	nearest	primary	primary- preferred	secondary
safe	28, 69, 89, 91, 92	80, 98, 98, 99, 99	74, 99, 99, 99, 99	0, 65, 83, 85, 88
normal	24, 68, 87, 89, 92	72, 99, 100, 100, 100	75, 98, 98, 98, 98	0, 69, 85, 89, 92
fsync_safe	28, 73, 87, 90, 90	68, 96, 98, 98, 98	78, 97, 98, 98, 98	0, 66, 80, 85, 86
replicas_safe	24, 74, 87, 88, 91	75, 98, 99, 99, 99	79, 98, 98, 98, 98	1, 67, 84, 87, 89
majority	26, 77, 91, 91, 92	73, 98, 99, 99, 99	77, 99, 99, 99, 100	0, 61, 82, 85, 89

TABLE III

CONSISTENCY: PERCENTAGE OF THE QUERIES STARTED AT 0, 2, 4 AND 8MS THAT WILL READ THE NEWEST INSERTED DATA WITH THE EACH ROW FOR A WRITE CONFIGURATION AND A COLUMN FOR THE READ CONFIGURATION. THE AVERAGE LATENCY ON A READ TRANSACTION IS FOR ALL READ QUERIES AROUND 1MS.

different systems. However multiple extensions can be made to this test method, next to testing more systems.

First of all, the strange behaviour of MongoDB and HBase of not allowing connections, could be researched in more detailed.

Secondly, the third element of CAP can be included: Partition tolerance. Right now the connection nodes are chosen when setting up the test, an extension could be to test all connections and see what happens in the case of a partition tolerance. How is the availability of each node, how long is a primary in MongoDB still accepting queries however it is cut off?

Thirdly, the availability and consistency tests could be tested together: when a node is shut down or cut off, is there the loose of data? Especially in MongoDB it could be the case that the data is already read on the primary but not yet replicated to the primary, what happens if the primary is cut off?

At last, the test parameters could be adapted, what happens with a different network infrastructure when the network distance is not equal any more, what happens with a higher and lower load? It could be possible in the end to have a mathematical formula which could predict the average consistency window for example.

## VII. RELATED WORK

The research towards database systems and their consistency guaranties is rare to have a measured approach. In recent paper (February 2014), Golab et al. states that there is only a limited amount of research done towards eventual consistency [8]. They present a new view on consistency, were already some research is done towards active analyse (how long before the data is replicated to all nodes), the amount of passive analyse is limited (what do the users see). Compared to the consistency results from this paper, both are discussed: as well the delays before the data is present everywhere but also on the acting of the specific systems, in example the behaviour of HBase.

Another extension of YCSB called YCSB++[9], provides more logging information on all systems in the first place, but they also test the consistency of HBase in regards of the client caches. The reasoning for this is that it is the standard run configuration of HBase, but the submitting of the client cache towards the system depends not only on the time, but also on the amount of traffic that is being submitted. In the study they compare different cache sizes and this shows already a difference in time. Furthermore, it is possible to disable this

caching in case there are records in the need of this strict consistency.

For availability benchmarking, there was no research found on related databases. However, research from 2004 [10] discuss a way to let a standalone system recover and provide a starting benchmark for it.

## VIII. CONCLUSION

A new testing method to quantify consistency and availability has been presented in this paper. With this method the effect of a stop of an instance can be measured and also the consistency window for different readers.

Another contribution are the results for two consistent and partition tolerant systems, HBase and MongoDB. These 2 systems are tested with the new method and show their relevance. According to the documentation, MongoDB and HBase both guarantee strict consistency but based on our results their characteristics are different: HBase blocks read queries till the completion of the write queries to guarantee a read will always be the same, MongoDB returns the ready query soon with new or old data before the write operation has been finished.

In the availability of the systems, HBase is in the standard configuration longer unavailable but with a change in the session time out, they can be made more the same. However both have a strange behaviour of not returning any results, in the case of HBase this is for a hard stop, in the case of MongoDB it is for a network partition. This doesn't happen in the other situations and in case of reconnecting to the database, the problems are solved.

## REFERENCES

- [1] E. A. Brewer, Towards robust distributed systems, in: PODC, 2000, p. 7.
- [2] E. Brewer, Cap twelve years later: How the "rules" have changed, Computer 45 (2) (2012) 23–29.
- [3] Hbase, apache hbase.  
URL <https://hbase.apache.org/>
- [4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, Bigtable: A distributed storage system for structured data, ACM Transactions on Computer Systems (TOCS) 26 (2) (2008) 4.
- [5] L. Hofhansl, Hbase: Acid in hbase (3 2012).  
URL <http://hadoop-hbase.blogspot.be/2012/03/acid-in-hbase.html>
- [6] The mongodb 2.6 manuel.  
URL <http://docs.mongodb.org/manual/>
- [7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with ycsb, in: Proceedings of the 1st ACM symposium on Cloud computing, ACM, 2010, pp. 143–154.

## **Bijlage F**

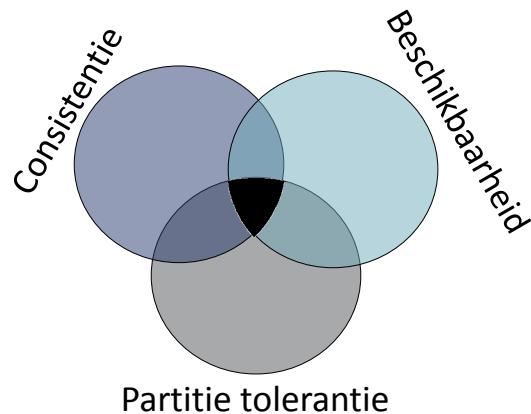
### **Poster**

De poster met als titel '*CAP in de praktijk: HBase and MongoDB*' bevindt zich op de volgende pagina.



## Het CAP Theorema

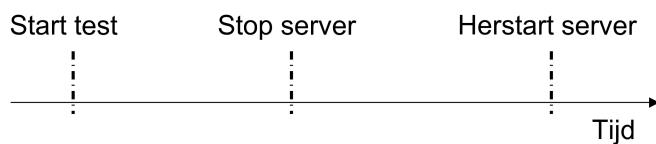
**Kies twee eigenschappen**



## MongoDB

- Document database systeem
- Replicatie met ReplicaSets
- Datadistributie met sharding
- 5 lees- en 5 schrijfconfiguraties
- Strikte consistentie bij standaard lees- en schrijfbewerking
- Partitie tolerant met beschikbaarheid voor meerderheid partitie

## Test methodiek

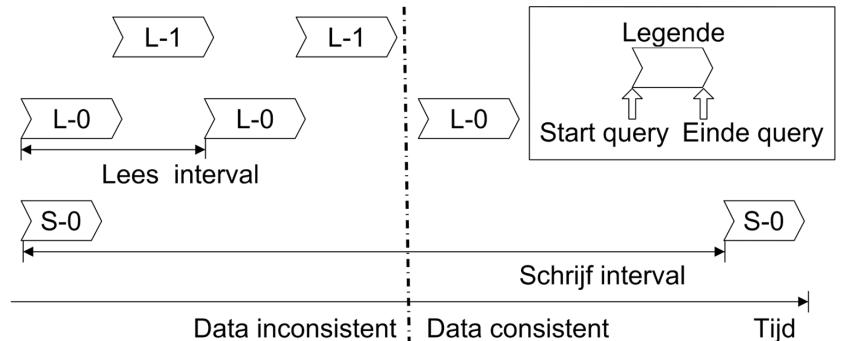


### Beschikbaarheid

- 300s: Stop server
- 600s: Herstart server

### Consistentie

- 1 schrijver
- Verschillende lezers
- Lees tot de data correct wordt gelezen



## Resultaten

### Beschikbaarheid

- **Stop van de service**
  - $\frac{1}{3}$  van de gevallen: onderbreking van enkele seconden
  - $\frac{2}{3}$  van de gevallen: geen effect

- **Netwerk onderbreking**

- Onderbreking: enkele seconden tot continue onderbreking.
- Opgelost bij opnieuw verbinden

- **Partitie <50% van servers**

- Onbeschikbaar voor schrijven en lezen

### Consistentie

- Schrijven
  - enkel garantie na operatie
- Lezen:
  - Onafhankelijk van gekozen schrijfoperatie
  - Kans op lezen van nieuwe waarde:

0 ms	2 ms	4 ms	6ms	
Primary	80%	98%	98%	99%
Secondary	0%	65%	83%	85%

# Bibliografie

- [1] David Bermbach en Stefan Tai. „Eventual consistency: How soon is eventual? An evaluation of Amazon S3’s consistency behavior”. In: *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. ACM. 2011, p. 1.
- [2] Kurt Bollacker e.a. „Freebase: a collaboratively created graph database for structuring human knowledge”. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, p. 1247–1250.
- [3] Dhruba Borthakur. „The hadoop distributed file system: Architecture and design”. In: *Hadoop Project Website* 11 (2007), p. 21.
- [4] Eric A. Brewer. „Towards Robust Distributed Systems (Abstract)”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC ’00. Portland, Oregon, USA: ACM, 2000, p. 7–. ISBN: 1-58113-183-6. DOI: [10.1145/343477.343502](https://doi.acm.org/10.1145/343477.343502). URL: <http://doi.acm.org/10.1145/343477.343502>.
- [5] Mike Burrows. „The Chubby lock service for loosely-coupled distributed systems”. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, p. 335–350.
- [6] Fay Chang e.a. „Bigtable: A distributed storage system for structured data”. In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), p. 4.
- [7] Edgar F Codd. „A relational model of data for large shared data banks”. In: *Communications of the ACM* 13.6 (1970), p. 377–387.
- [8] Brian F Cooper e.a. „Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, p. 143–154.
- [9] *DB-Engines Ranking - popularity ranking of database management systems*. URL: <http://db-engines.com/en/ranking> (bezocht op 20-07-2014).
- [10] Harm De Weirdt. „Configuratieafhankelijkheden gebruiken om gedistribueerde applicaties efficiënt te beheren in een hybride cloud.” KU Leuven, 2014.
- [11] Jeffrey Dean en Sanjay Ghemawat. „MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), p. 107–113.

- [12] Ramez Elmasri en Shamkant Navathe. *Fundamentals of Database Systems*. 6th. USA: Addison-Wesley Publishing Company, 2010. ISBN: 0136086209, 9780136086208.
- [13] Lars George. *HBase: the definitive guide*. "O'Reilly Media, Inc.", 2011.
- [14] Sanjay Ghemawat, Howard Gobioff en Shun-Tak Leung. „The Google file system”. In: *ACM SIGOPS Operating Systems Review*. Deel 37. 5. ACM. 2003, p. 29–43.
- [15] Wojciech Golab e.a. „Eventually consistent: not what you were expecting?” In: *Communications of the ACM* 57.3 (2014), p. 38–44.
- [16] Lars Hofhansl. *HBase: Acid in HBase*. Mrt 2012. URL: <http://hadoop-hbase.blogspot.be/2012/03/acid-in-hbase.html> (bezocht op 10-07-2014).
- [17] J Hugg. *Key-value benchmarking*. 2010. URL: <http://voltdb.com/blog/voltdb-benchmarks/key-value-benchmarking/> (bezocht op 06-07-2014).
- [18] Patrick Hunt e.a. „ZooKeeper: Wait-free Coordination for Internet-scale Systems.” In: *USENIX Annual Technical Conference*. Deel 8. 2010, p. 9.
- [19] ZikaiWang James Chin. *HBase: A Comprehensive Introduction*. 2011. URL: <http://cs.brown.edu/courses/cs227/archives/2011/slides/mar14-hbase.pdf> (bezocht op 10-07-2014).
- [20] Christos Kalantzis. *A Netflix Experiment: Eventual Consistency != Hopeful Consistency*. Planet Cassandra. 2013. URL: <http://planetcassandra.org/blog/post/a-netflix-experiment-eventual-consistency-hopeful-consistency-by-christos-kalantzis/> (bezocht op 06-07-2014).
- [21] Avinash Lakshman en Prashant Malik. „Cassandra: A Decentralized Structured Storage System”. In: *SIGOPS Oper. Syst. Rev.* 44.2 (apr 2010), p. 35–40. ISSN: 0163-5980. DOI: [10.1145/1773912.1773922](https://doi.org/10.1145/1773912.1773922). URL: <http://doi.acm.org/10.1145/1773912.1773922>.
- [22] Todd Lipcon. „Design Patterns for Distributed Non-Relational Databases”. In: *Design Patterns for Distributed Non-Relational Databases* (2009).
- [23] Cary Millsap. *Optimizing Oracle Performance*. "O'Reilly Media, Inc.", 2003.
- [24] *MongoDB Concurrency*. URL: <http://docs.mongodb.org/manual/faq/concurrency/> (bezocht op 10-07-2014).
- [25] *MongoDB Manual*. URL: <http://docs.mongodb.org/manual/> (bezocht op 10-07-2014).
- [26] *MongoDB: Replication Introduction*. URL: <http://docs.mongodb.org/manual/core/replication-introduction/> (bezocht op 10-07-2014).
- [27] *MongoDB: Sharding Introduction*. URL: <http://docs.mongodb.org/manual/core/sharding-introduction/> (bezocht op 10-07-2014).
- [28] Swapnil Patil e.a. „YCSB++: benchmarking and performance debugging advanced features in scalable table stores”. In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM. 2011, p. 9.

- [29] *Pgpool-II: User manuel*. URL: <http://www.pgpool.net/docs/latest/pgpool-en.html> (bezocht op 18-07-2014).
- [30] Pouria Pirzadeh, Junichi Tatenuma en Hakan Hacigumus. „Performance evaluation of range queries in key value stores”. In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE. 2011, p. 1092–1101.
- [31] A. Popescu. *NoSQL benchmarks and performance evaluations*. 2010. URL: <http://nosql.mypopescu.com/post/734816227/nosql-benchmarks-and-performance-evaluations> (bezocht op 06-07-2014).
- [32] Alex Popescu. *Presentation: NoSQL at CodeMash – An Interesting NoSQL categorization*. Feb 2010. URL: <http://nosql.mypopescu.com/post/396337069/presentation-nosql-codemash-an-interesting-nosql> (bezocht op 03-02-2014).
- [33] Postgresql. *PostgreSQL - Replication, Clustering, and Connection Pooling*. Okt 2013. URL: [http://wiki.postgresql.org/wiki/Replication,\\_Clustering,\\_and\\_Connection\\_Pooling](http://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling) (bezocht op 03-02-2014).
- [34] Tilmann Rabl e.a. „Solving big data challenges for enterprise application performance management”. In: *Proceedings of the VLDB Endowment* 5.12 (2012), p. 1724–1735.
- [35] Arnaud Schoonjans. „Een critische evaluatie van beschikbaarheid in gedistribueerde opslag systemen”. KU Leuven, 2014.
- [36] Ben Scofield. *NoSQL – Death to Relational Databases(?)* Jan 2010. URL: <http://www.slideshare.net/bscofield/nosql-codemash-2010> (bezocht op 03-02-2014).
- [37] Christof Strauch. *NoSQL Databases*. 2010. URL: <http://www.christof-strauch.de/nosqldb.pdf>.
- [38] *The Apache HBase Reference Guide*. URL: <http://hbase.apache.org/book/book.html> (bezocht op 18-07-2014).
- [39] Bogdan George Tudorica en Cristian Bucur. „A comparison between several NoSQL databases with comments and notes”. In: *Roedunet International Conference (RoEduNet), 2011 10th*. IEEE. 2011, p. 1–5.
- [40] Bart Vanbrabant. „A Framework for Integrated Configuration Management of Distributed Systems (Een raamwerk voor geïntegreerd configuratiebeheer van gedistribueerde systemen)”. Proefschrift. Jun 2014. URL: <https://lirias.kuleuven.be/handle/123456789/453199>.
- [41] Hiroshi Wada e.a. „Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers’ Perspective.” In: *CIDR*. Deel 11. 2011, p. 134–143.

## Fiche masterproef

*Student:* Thomas Uyttendaele

*Titel:* Consistentie en beschikbaarheidstesten voor gedistribueerde database systemen

*Engelse titel:* Consistency and availability benchmarking for distributed database systems

*UDC:* 681.3

*Korte inhoud:*

De meest gebruikte database management systemen zijn momenteel relationele of NoSQL systemen waarbij er verschillen zijn naar het ondersteunde datamodel. Daarnaast zijn er ook verschillende keuze gemaakt naar het opzetten van deze systemen in een gedistribueerde omgeving.

Volgens het CAP theorema, kan een systeem niet tegelijk consistentie en hoge beschikbaarheid aanbieden en partitie tolerant te zijn. Deze thesis introduceert een model om verschillende database systemen gekwantificeerd te vergelijken naar consistentie en beschikbaarheid. Bij consistentie wordt het leesgedrag vergeleken bij het invoegen of aanpassen van data: hoelang het duurt vooraleer de data beschikbaar is voor elke gebruiker en het gedrag van de query gedurende deze periode.

Voor de beschikbaarheidstesten wordt het gedrag van de queries opgevolgd terwijl database instantie stoppen en of er netwerk partities ontstaan. Dit model is praktisch uitgewerkt met behulp van YCSB en verschillende database systemen zijn getest.

Waar zowel HBase als MongoDB stikte consistentie afleveren, is er een praktisch verschil in hun gedrag: bij HBase worden leesbewerkingen, gestart na de schrijfbewerking, uitgesteld om de nieuwe data terug te geven onmiddellijk na het voltooien van de schrijfbewerking. Bij MongoDB daarin tegen kunnen leesbewerkingen de nieuwe data al teruggeven vooraleer de schrijfbewerking is voltooid. Ook zijn er verschillend garanties bij MongoDB mogelijk voor de lees- en schrijfbewerkingen.

Bij de beschikbaarheidstesten zijn er verschillen tussen de aanpak bij MongoDB, HBase en Pgpool-II zowel wanneer een server onbeschikbaar of terug beschikbaar wordt.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Gedistribueerde systemen

*Promotor:* Prof. dr. ir. Wouter Joosen

*Assessor:* Prof. dr. ir. Tias Guns,  
Prof. dr. ir. Christophe Huygens

*Begeleider:* Dr. ir. Bart Vanbrabant  
Dr. Bert Lagaisse