

Evaluatie van consistentie en beschikbaarheid in 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

Assessoren:
Dr. Tias Guns
Dr. ir. Christophe Huygens

Begeleiders:
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.

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 vele vruchtbare gesprekken gehad over de mogelijke testmethodes en onze kennis over de verschillende database systemen kunnen delen.

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

Thomas Uyttendaele

Inhoudsopgave

Voorwoord	i
Inhoudsopgave	ii
Samenvatting	iv
1 Inleiding	1
1.1 Relationale en NoSQL DBMS's	2
1.2 Vergelijking van DBMS's naar performantie en CAP	6
1.3 Doelstelling en bijdrage	10
1.4 Verder verloop	11
2 Voorgestelde testmethodiek	13
2.1 Stap 1: Opstellen van de testomgeving	14
2.2 Stap 2: Kalibratie van de testomgeving	14
2.3 Stap 3: Testen van de systemen	16
2.4 Stap 4: Verzamelen en analyseren van de testdata	20
2.5 Conclusie	21
3 Implementatie	23
3.1 Selectie van de DBMS's	23
3.2 Gedetailleerde bespreking van de model DBMS's	24
3.3 Selectie en uitwerking van de testsoftware	31
3.4 Installatie en opstelling van de DBMS's en YCSB	32
3.5 Uitvoeren van de kalibratie en testen	34
3.6 Verzamelen en analyse van de testresultaten	37
3.7 Conclusie	37
4 Observaties	39
4.1 Kalibratie	39
4.2 Beschikbaarheidstest	44
4.3 Consistentietest	48
4.4 Conclusie	55
5 Analyse van de resultaten	57
5.1 Kalibratie	57
5.2 Beschikbaarheidstest	58
5.3 Consistentietest	61
5.4 Conclusie	65

6 Conclusie	67
6.1 Uitdagingen	67
6.2 Verder werk	68
6.3 Evaluatie van de doelstellingen en bijdragen	69
A Besprekking van verschillende DBMS's	73
A.1 Column database	73
A.2 Document database	74
A.3 Key-Value database	75
A.4 Relationale database	77
B Overzicht van gedetailleerde implementatie keuzes	79
C Extern beschikbare code en resultaten	83
D Paper: "CAP in practice: HBase and MongoDB"	85
E Poster: "CAP in de praktijk: MongoDB"	93
Bibliografie	95

Samenvatting

In de database wereld zijn er momenteel twee grote categorieën van database systemen actief: de relationele en NoSQL systemen. De laatste categorie maakt verschillende keuzes in het CAP theorema en heeft uit een grote diversiteit aan datamodellen en performantie-eigenschappen.

Deze thesis focust op de uitwerking van een benchmarking tool om consistentie en beschikbaarheid in een gedistribueerde omgeving te testen.

De beschikbaarheidstesten onderzoeken het gedrag van het gehele database systeem bij het in- en uitschakelen van een service, een node in het gedistribueerde opslag systeem of netwerk verbinding. De consistentietest bestudeert hoe het systeem omgaat met gelijktijdige schrijf- en leesbewerkingen op hetzelfde datarecord.

Deze benchmark onderzoekt het gedrag van een database systeem in de praktijk en vergelijkt dit met de documentatie. Daarnaast worden verschillende database systemen met elkaar vergeleken.

Drie verschillende database systemen worden getest met behulp van de benchmarking tool, waarna de resultaten geanalyseerd worden. Op basis van deze resultaten komt verschillend gedrag tussen de systemen tot uiting zoals op een verschillende wijze omgaan met gelijktijdig lezen en schrijven van een data-element.

De installatie en configuratie van de benchmarking tool en de reeds geteste database systemen is geautomatiseerd om te kunnen opstellen en uitvoeren zonder voorkennis van de specifieke systemen. Met een beperkte kennis en kost kan de tool uitgevoerd worden op andere database systemen.

Hoofdstuk 1

Inleiding

De hedendaagse meest gebruikte database management systemen (DBMS's) zijn relationele of NoSQL DBMS's [9]. De NoSQL systemen bestaan uit een waaier van systemen met verschillen in datamodel, performantie, beschikbaarheid of consistentie. Het CAP theorema van Erik Brewer[4] stelt dat elk gedistribueerd dataopslag niet tegelijk onmiddellijke consistentie, hoge beschikbaarheid kan zijn en partitie tolerant kan ondersteunen. Elk DBMS maakt zijn eigen keuze en afweging tussen de verschillende garanties en eigenschappen die het levert.

Deze thesis beschrijft een testmethode om DBMS's te vergelijken op basis van de consistentie- en beschikbaarheidsgaranties. De methode onderzoekt het gedrag van het volledige DBMS bij het in- en uitschakelen van een node in het gedistribueerde opslag systeem of netwerkverbinding. Daarnaast wordt bestudeerd hoe de data van een schrijfbewerking zichtbaar wordt voor gelijktijdige leesbewerkingen. Deze testen worden gebruikt om het theoretische beschreven gedrag te vergelijken met de praktijk. Daarnaast kunnen de verschillen tussen DBMS's vergeleken worden.

De testmethode wordt toegepast op drie opslag systemen die strikte consistentie garanderen en partitie tolerant zijn: HBase en MongoDB zijn NoSQL systemen, PgPool-II (PostgreSQL) is een relationeel systeem. Uit de resultaten blijkt dat PgPool-II op een andere wijze de beschikbaarheid van de verschillende systemen registreert dan de 2 andere systemen. HBase en MongoDB behandelen gelijktijdige lees- en schrijfbewerking op dezelfde data verschillend. HBase zal leesbewerkingen uitstellen tot een volledige voltooiing van de schrijfbewerking, MongoDB zal de leesbewerkingen al nieuwe data laten lezen voor het volledig voltooien van de schrijfbewerking.

De volgende sectie behandelt een besprekings van relationele en NoSQL DBMS's, gevolgd door een overzicht van de testmethodes en resultaten van andere studies. Hieruit blijkt dat er een afwezigheid is van testmethodes naar consistentie en beschikbaarheid. Daarna zullen de doelstellingen en bijdrage van de thesis geformuleerd worden. Tenslotte wordt een overzicht gegeven van de rest van de thesis.

1. INLEIDING

1.1 Relationale en NoSQL DBMS's

Zoals voordien vermeldt, zijn de populaire DBMS's momenteel de relationele en NoSQL systemen. In deze sectie zullen beide categorieën besproken worden, eerst komt het relationele DBMS aan bod, gevolgd door NoSQL.

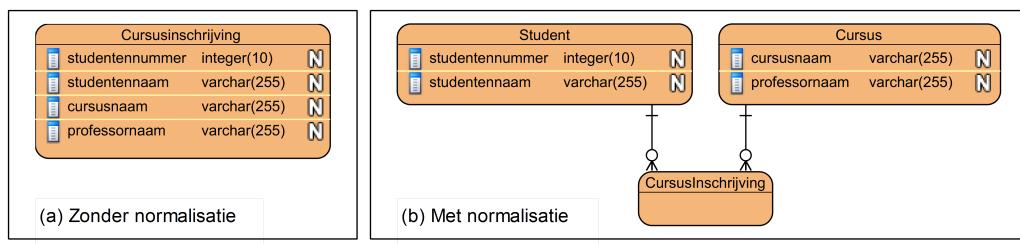
1.1.1 Relationele database

Het RDBMS is gebaseerd op het relationele model voor het structureren van de data in een database. Dit model is uitgebreid besproken in het artikel van E. Codd [7]. Voorbeelden van hedendaagse populaire relationele DBMS's (RDBMS's) zijn Oracle, MySQL en PostgreSQL.

Het relationele model vertrekt van theoretische wiskundige principes zoals de set-theorie en de eerste-orde predicaten logica. Het model organiseert de data in tabellen en legt relaties tussen de tabellen. De tabel heeft kolommen die verschillende velden voorstellen waarbij elke rij een collectie van gerelateerde datawaarden is. De relaties tussen de verschillende tabellen tonen 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, dezelfde gedachtegang is toepasbaar op 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: Relationale 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) Of een database transactie moet volledig uitgevoerd worden, of er heeft geen enkele bewerking plaatsgevonden.

Consistent (Consistency) Een transactie behoudt consistentie als de volledige uitvoering van de transactie de database van een consistente staat naar een andere consistentie staat brengt. Een consistente staat is een staat die ervoor zorgt dat waardes 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 geld.

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. Daartegenover staat wel dat dit de complexiteit van de RDBMS groeit, ook indien dit voor bepaalde toepassingen misschien niet nodig is. Een moeilijkheid is het schalen van het systeem bij een groter wordende dataset; door het toepassen van normalisatie dient er bij een enkele query data van verschillende tabellen gecombineerd te worden. Dit zorgt ervoor dat er in eerste instantie verticaal geschaald zal worden, het gebruiken van krachtigere hardware. Het toevoegen van extra servers, horizontaal schalen, is in veel gevallen slechts beperkt ondersteund.

1.1.2 NoSQL database

Dit hoofdstuk is gebaseerd op het boek NoSQL Databases[37].

NoSQL DBMS zijn ontstaan door groei en globalisering van de computersystemen en de bijhorende databases. Een RDBMS is gebouwd met een 'one size fits all'-gedachte, maar deze systemen veroorzaken 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 in vergelijking 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 mag dit even duren.
- **Hogere doorvoer:** Talrijke NoSQL systemen bieden een hogere doorvoer van data aan. In veel gevallen is dit een gevolg 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

1. INLEIDING

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 krachtigere 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.

- **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 nood aan de vertaling van een object naar een databasestructuur. Bij het ontwerp van NoSQL systemen 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 schrijf-acties 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.
- Uiteindelijke 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 van een verschillend dataelement 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 het CAP theorema van Erik Brewer[4]. CAP zegt dat een gedistribueerd systeem maar twee van de drie 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 één dataopslag 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]

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]

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]

1. INLEIDING

1.1.3 Bespreking van verschillende DBMS's

Voorbeeldsystemen uit de relationele systemen en 3 categorieën van NoSQL komen verder aan bod. 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
- 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 maken ook keuzes uit performantie en CAP theorema, deze keuzes hebben hun invloed op de prestaties en het gedrag voor de gebruikers. In deze sectie zal er gekeken worden welke methodes er reeds beschikbaar zijn voor het analytisch vergelijken van de performantie, consistentie en beschikbaarheid op basis van het gedrag in de praktijk. Daarnaast worden al mogelijke resultaten kort besproken.

1.2.1 Performantie benchmarking

Voor de vergelijking van de performantie van verschillende DBMS's bestaan er reeds enkele tools en studies. Een blogpost van A. Popescu [31] geeft een overzicht van verschillende benchmarking tools.

Vele DBMS's hebben **interne benchmarking tools**, waarmee de database op verschillende configuraties kunnen getest en vergeleken worden. Deze resultaten zijn nuttig na het kiezen van een DBMS. Het systeem kan getest worden met variërende

1.2. Vergelijking van DBMS's naar performantie en CAP

parameters en kan helpen met het uitzoeken wat de bottleneck is van een bepaald systeem. Een voorbeeld hiervan is mongoperf¹ 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 een specifieke categorie van systemen, beide met een focus op het publiceren van de resultaten. Elke van deze studies brengt nieuwe kennis van de systemen maar heeft ook zijn beperkingen. Het totaal pakket 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 de testen opnieuw kunnen uitvoeren met de systemen in hun configuratie. Het is namelijk niet gegarandeerd dat het resultaat van een jaar geleden overeenstemmend 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 testen zolang een basisset van 5 queries ondersteund wordt: het invoegen, updaten, verwijderen, opvragen van een enkel record en 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. Een update kan bijvoorbeeld 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

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

1. INLEIDING

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?

Een recent artikel [15] (maart 2014), stelt dat er momenteel nauwelijks gekwantificeerde methodes bestaan om de uiteindelijke 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 meet 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 record een 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 uiteindelijk consistent door het gebruiken van buffers bij de gebruiker. Vervolgens testen zij hoe lang het duurt voor de data ook kan gelezen 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 uiteindelijke consistentie voor Cassandra en MongoDB, zijn de resultaten niet beschikbaar in het artikel of op de website.

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

Hoe gaan deze systemen om met 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 dit 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.

Na een 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.

1.2.4 Probleemstelling

Uit het onderzoek blijkt dat er al verschillende tools zijn voor het vergelijken van de performantie van DBMS's, ook zijn er al de nodige vergelijkende studies uitgevoerd.

1. INLEIDING

De situatie is anders als er gekeken worden naar consistentie- en beschikbaarheidstesten. Er zijn slechts enkele tools en resultaten beschikbaar welke voornamelijk focussen wanneer de data beschikbaar is op elke servers.

Indien een ontwikkelaar wil weten welke consistentie- en beschikbaarheidsgaranties een systeem heeft, dient de ontwikkelaar nu te vertrouwen op de documentatie of zelf testen uit te voeren. Na het lezen van de documentatie kan afgeleid worden wat er gegarandeerd worden maar niet hoe het systeem dit realiseert. Om het gedrag in de praktijk te onderzoeken kan de ontwikkelaar de broncode bestuderen of zelf de testen uitvoeren. Voor het uitvoeren van testen dient het systeem eerst opgezet worden waarna de testen eerst ontwikkeld of aangepast worden en uitgevoerd worden. Beide pistes kosten tijd en moeite waardoor dit slechts voor een beperkt aantal systemen uitvoerbaar is.

Momenteel is er kloof tussen wat een ontwikkelaar kan gebruiken om de DBMS's te vergelijken en wat hij nodig heeft om een volledig beeld te krijgen, in deze thesis zal er gepoogd deze kloof te dichten.

1.3 Doelstelling en bijdrage

Deze thesis zal op basis van drie verschillende doelstellingen streven om de kloof te verkleinen tussen wat nodig en wat beschikbaar is als informatie voor de selectie en configuratie van een DBMS.

Het *eerste* doel van deze thesis is om een **benchmarking tool te ontwikkelen** die een variëteit aan systemen ondersteunt. Deze testen zullen het gedrag onderzoeken bij het uitvallen van een service, een node in het gedistribueerde opslag systeem of netwerkverbinding, gegroepeerd zijn dit de beschikbaarheidstesten. Een tweede categorie zijn de consistentietesten, hierbij wordt getest hoe nieuw geschreven data gelezen wordt door dezelfde gebruiker en anderen. Er wordt niet gefocust op het tijdstip dat de data beschikbaar op verschillende servers, maar er wordt gekeken vanuit het gebruikersperspectief: wanneer leest deze de oude of nieuwe data lezen, hoe beïnvloedt een schrijfbewerking een gelijktijdige leesbewerking op dezelfde data, ...

Het *tweede* doel is het **analyseren van verschillende systemen** met behulp van de ontwikkelde benchmarking tool. In deze thesis zal de tool uitgevoerd worden op drie voorbeeldsystemen: HBase, MongoDB en Pgpool-II(uitbreiding van PostgreSQL). Als resultaat kunnen er conclusies getrokken worden over het gedrag in de praktijk bij beschikbaarheid en consistentie. Dit wordt vergeleken met de garanties en eigenschappen uit de documentatie. Daarnaast toont dit ook aan dat de ontwikkelde tool geschikt is voor het testen van database systemen voor consistentie en beschikbaarheid.

Het *derde* en laatste doel is om voor het **eenvoudig reproduceren van de uitgevoerde testen** te zorgen, dit geldt voor de installatie en configuratie van de DBMS's als voor de benchmarking tool. Op deze manier kunnen de testen eenvoudig opnieuw uitgevoerd worden op een andere infrastructuur, met een nieuwe versie van een DBMS of als verificatie van de testen op dezelfde structuur. Daarnaast is het ook mogelijk om met beperkte moeite de benchmarking tool uit te breiden voor nieuwe DBMS's.

Met behulp van deze drie doelstellingen brengt de thesis een ontwikkelaar verschillende nieuwe elementen bij. Allereerst is het voor toekomstige ontwikkelaars mogelijk om voor de drie onderzochte systemen te begrijpen hoe deze consistentie en beschikbaarheid aanbieden. Pgpool-II heeft bijvoorbeeld een server die geen data opslaat maar enkel als router dient. Dit zorgt ervoor dat deze op een gecentraliseerde wijze de status van de verschillende servers kan bijhouden. HBase en MongoDB hebben een meer gedecentraliseerde aanpak en kiezen zelf een leider. Deze tweede methode zorgt ervoor dat bij het falen van de leider het database systeem verder kan werken. Als bij Pgpool-II de leider faalt, is het volledige systeem niet beschikbaar in de eenvoudigste configuratie.

Daarnaast kunnen de testen opnieuw uitgevoerd worden, maar ook aangepast en uitgebreid worden. Dit is zowel het toevoegen van nieuwe DBMS's als het aanpassen van de testen. Met deze mogelijkheden kunnen ontwikkelaars hun eigen DBMS en infrastructuur testen onder omstandigheden die voor hen het meest toepasbaar zijn. Hiermee komen ze te weten hoe het systeem zich zal gedragen in bepaalde scenario's en kunnen hiermee rekening houden bij het ontwikkelen van de software.

Dit onderzoek neemt tijd in beslag van de ontwikkelaar maar dit zorgt ervoor dat hij allereerst een systeem kan kiezen dat het gewenste gedrag heeft. De kennis van de eigenschappen en het gedrag zorgt ervoor dat de implementatie sneller en efficiënter kan gebeuren door het optimaal gebruik van de garanties van het DBMS.

1.4 Verder verloop

Deze thesis behandelt het ontwikkelen en uitvoeren van een benchmark tool om consistentie en beschikbaarheid te testen op relationele en NoSQL databases. Met behulp van de testmethode en de resultaten kan de selectie van een DBMS beter onderbouwd worden. Deze thesis zal in de volgende hoofdstukken de verschillende bouwblokken verder bespreken.

Hoofdstuk 2: Voorgesteld testmethode Dit hoofdstuk bespreekt een theoretische methode voor het testen van de consistentie en beschikbaarheid van een DBMS. Deze methode is implementatie onafhankelijk en beschrijft op hoog niveau de verschillende stappen van de testmethode.

1. INLEIDING

Hoofdstuk 3: Implementatie Dit hoofdstuk bespreekt de vertaling van de theoretische testmethode naar een volledig werkend systeem. Het hoofdstuk behandelt onder andere de selectie en de werking van de voorbeeld DBMS's, de keuzes en configuratie van de benchmarking tool en DBMS's en eindigt met een uitleg hoe de testen in de praktijk uitgevoerd worden.

Hoofdstuk 4: Observaties Dit hoofdstuk presenteert de testdata van de verschillende voorbeeld DBMS's. Verschillende reacties van eenzelfde systeem op een bepaalde test komen aan bod en bepaalde reacties worden op een grafische wijze voorgesteld.

Hoofdstuk 5: Analyse van de resultaten De resultaten van de testen worden in dit hoofdstuk ontleed en geanalyseerd, met behulp van de documentatie van elk systeem wordt dan de reden achterhaald. Tenslotte worden de verschillende systemen met elkaar vergeleken.

Hoofdstuk 6: Conclusie Dit hoofdstuk sluit de thesis af met een samenvatting en de bijdragen van het werk, mogelijke uitbreidingen en een overzicht van de uitdagingen die de thesis met zich meebracht.

Appendix A: Bespreking van verschillende DBMS's Deze appendix bespreekt verschillende database management systemen in meer detail.

Appendix B: Overzicht van gedetailleerde implementatie keuzes Deze appendix bevat een gedetailleerde bespreking van testparameters.

Appendix C: Extern beschikbare code en resultaten Deze appendix bevat een overzicht van de code die in verband met deze thesis werden ontworpen en de resultaten die werden gegenereerd.

Appendix D: Paper Deze appendix bevat de paper met een bespreking van het CAP theorema voor HBase en MongoDB in de praktijk.

Appendix E: Poster Deze appendix bevat een poster over MongoDB en het CAP theorema in de praktijk.

Hoofdstuk 2

Voorgestelde testmethodiek

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

Opstellen van de testomgeving De eerste stap bestaat uit 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.

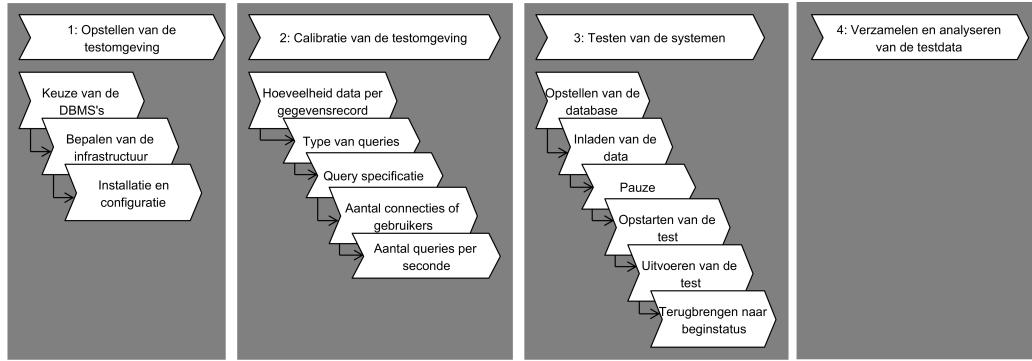
Kalibratie van de testomgeving In de uiteindelijke testen wordt het gedrag onder matige belasting getest. Afhankelijk van de gekozen systemen en de netwerkinfrastructuur zal dit voor elke DBMS een verschillende hoeveelheid bewerkingen geven. Deze stap bepaalt welke queries er uitgevoerd worden, hoeveel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

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 bewerking zich gedraagt voor, tijdens en na het falen en herstellen van een systeem. Voor de consistentie is het mogelijk om zowel een 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 visueel voorgesteld. Met behulp van de testdata, is het mogelijk om bepaalde conclusies te maken over de beschikbaarheids- en consistentiegaranties van de verschillende DBMS's.

In de volgende secties worden de verschillende stappen uitgewerkt.

2. VOORGESTELDE TESTMETHODIEK



Figuur 2.1: Een gedetailleerd overzicht van het testproces met al de verschillende stappen

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 configuraties of verschillende configuraties, er kunnen ook verscheidene database systemen naast elkaar besproken worden.

Bepalen van de infrastructuur Vervolgens gebeurt de keuze van de infrastructuur. Dit gebeurt door het aantal instanties per systeem vast te leggen en de hardware te kiezen.

Installatie en configuratie Het lokaal installeren en configureren van een softwarepakket 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.

Deze gedistribueerde configuratie stap staat in voor het in contact brengen van de verschillende services op de verschillende servers. Voor verschillende services zijn er andere configuratie methodes, bij sommigen gebeurt dit door middel van configuratiebestanden voor het opstarten van de service. Bij andere gebeurt dit via de API na het opstarten van de services.

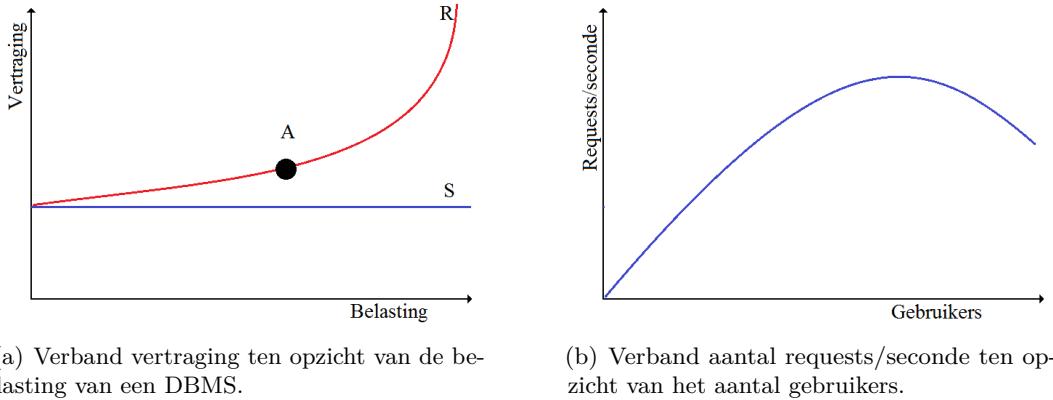
Na het uitvoeren van deze stap, zou het DBMS moeten werken zoals vereist.

2.2 Stap 2: Kalibratie van de testomgeving

Verschillende DBMS's ondersteunen een andere belasting aan, ook bij een zelfde DBMS zijn er grote verschillen bij het veranderen van de configuratie of de infrastructuur. Het de bedoeling dat alle database systeem een matige belasting te

2.2. Stap 2: Kalibratie van de testomgeving

hebben bij de testen, op deze manier kan er gefocust worden op de consistentie en beschikbaarheid. Bij een hoge belasting wordt er meer gefocust op de performantie en is er een hogere vertraging. 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 kalibratie

De belasting van een database is afhankelijk van verschillende parameters. In dit gedeelte wordt er gefocust op 5 groepen die in de benchmark tool geconfigureerd worden:

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

Type van queries De opgeslagen data kan opgevraagd worden op verschillende wijze. Data kan ingevoegd, aangepast, opgevraagd of verwijderd worden. Daarbij kan dit telkens gebeuren voor 1 of meerdere records tegelijk. Afhankelijk van de relatieve verhouding, is er een andere belasting en vertraging: sommige DBMS's zijn meer geschikt voor een dominantie in leesacties, andere prefereren een dominantie in schrijffacties. Enkele systemen lezen data efficiënt in grote aantallen, andere lezen goed in kleine aantallen.

Query specificatie Bij het opvragen of verwijderen van een record, kan er een verschil zijn naar processing tijd afhankelijk hoe lang geleden de record geschreven of gelezen is en of naburige data onlangs gelezen is. Dit kan het geval zijn door caching methodes die het lezen van bepaalde records versnellen. Vandaar dat de distributie waarmee verschillende record gelezen en geschreven gekozen dient te worden. Voorbeelden van verschillende technieken zijn: voornamelijk de laatste data lezen, een uniforme kans voor alle data of bepaalde records regelmatig lezen.

2. VOORGESTELDE TESTMETHODIEK

Aantal connecties of gebruikers Bij een gedistribueerde database systeem kunnen er meerdere gebruikers 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: $\#Queries = \#Gebruikers * \#QueriesPerGebruiker$. Bij het uitvoeren van deze test wordt er verondersteld dat elke 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 van het aantal gebruikers versus het aantal bewerkingen per second zal een gelijkaardig verloop hebben als de grafiek in figuur 2.2(b).

Aantal queries per seconde In de vorige stap is er het optimale aantal gebruikers bepaald om het maximaal aantal bewerkingen uit te voeren. In het begin is er gesteld dat er gezocht wordt naar een matige belasting. Deze matige belasting komt overeen met punt A uit figuur 2.2(a). Op dit moment is er nog een lineaire verhouding tussen de wachttijd en de belasting.

Met de parameters afkomstig uit de kalibratie, 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 namelijk de beschikbaarheid en consistentietesten, welke dezelfde 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 structuur van de data. Deze structuur wordt zo goed mogelijk meegegeven aan het DBMS zodat deze aan optimale schijfallocatie kan doen en de bewerkingen ook kan optimaliseren. Indien mogelijk dient ook meegegeven worden hoe de data opgevraagd zal worden, meestal gebeurt dit door middel van een bepaald veld die als sleutel zal gebruikt worden.

Inladen van de data Een bepaalde hoeveel data wordt vooraf ingeladen. Dit wordt gedaan om een basishoeveelheid data te hebben die nodig is voor de verdere initialisatie van de database. Verschillende DBMS's verdelen van de data over verschillende servers, dit wordt sharding genoemd. In bepaalde DBMS's gebeurt deze sharding pas bij het overschrijven van een bepaalde hoeveelheid records, om deze reden wordt er data ingeladen zodat deze sharding al is toegepast voor het opstarten van de eigenlijke testen. 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 schrijven van de data. Dit kan onder andere te wijten zijn doordat data nog weggeschreven moet worden naar schijf, gerepliceerd over verschillende servers of in bepaalde systemen zou het kunnen dat de sharding gebeurt op momenten met weinig belasting. Met het toevoegen van de wachtpériode wordt deze verhoogde belasting van de eigenlijke test weggehouden.

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 dan het gemiddelde. Deze hogere tijd is onder andere te verklaren door de connecties die opgezet moet worden en caches voor gelezen data die worden gevuld. Soms is deze lager omdat de schijf nog niet belast is of omdat er nog veel schrijfbuffers leeg zijn. Om dit gedrag te vermijden, wordt de data, verzameld in 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 data en de database worden volledig verwijderd. Belangrijk in dit geval is het controleren of de data volledig verwijderd is, in bepaalde gevallen wordt de data enkel onzichtbaar gemaakt bij het verwijderen, hetgeen kan een volgende test kunnen beïnvloeden.

De twee verschillende testmethodes zullen nu in 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 oorzaken getest die de systemen onbeschikbaar maken, terwijl de belasting uit de kalibratiestap wordt toegepast.

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

Harde stop De DMBS service wordt onmiddellijk gestopt door het proces 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 pas later opmerken en hij de service opnieuw opstart.

2. VOORGESTELDE TESTMETHODIEK

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 DBMS kan sterk verschillend reageren op de verschillende situaties: de eerste situatie is het eenvoudigst te behandelen aangezien het systeem de andere op de hoogte kan brengen. In de tweede situatie wordt het moeilijker omdat de andere systemen niet op de hoogte kunnen gebracht worden. De systemen krijgen bij het contacteren van de service het antwoord dat deze onbereikbaar 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, of er een hoge vertraging op de netwerk verbinding zit.

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

2.3.2 Consistentietest

In de consistentietest wordt onderzocht welke consistentie-eigenschappen het DBMS garandeert. Zoals voordien besproken in deel 1.1.2, bestaan er verschillende soorten van consistentie.

In deze testen is er gekozen om caching bij de gebruiker **uit te schakelen**. Dit is gebeurt omdat dat dit gedrag onvoorspelbaar is en afhankelijk van andere acties op de connectie. Een andere reden is dat uiteindelijke consistentie alleen een probleem is voor data die onmiddellijk beschikbaar moet zijn, met andere woorden data die men niet mag cachen. Na het uitschakelen van de caching kan het zijn dat de belasting op de server hoger is.

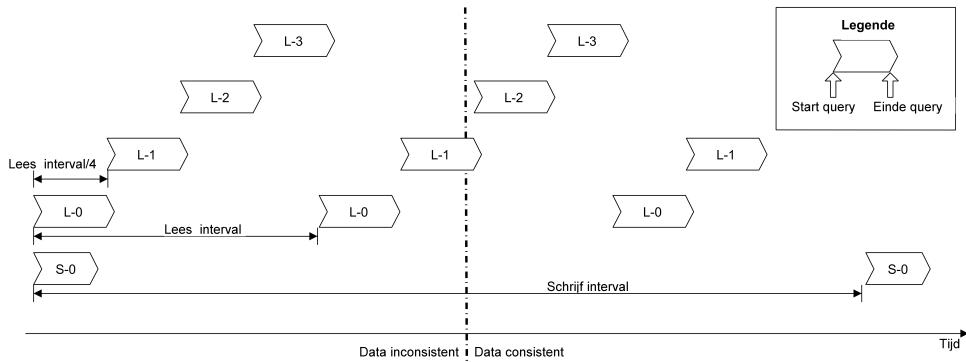
Beschrijving van de test Deze test bestaat uit 3 soorten gebruikers: er is een gebruiker die data schrijft (=S), een aantal lezers (=L's) en tenslotte zijn er nog andere gebruikers die zorgen voor de basisbelasting. De berekening van deze basisbelasting komt verder aan bod. Het is belangrijk dat er een exakte 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 voorschrijft, voorafbepaalde data weg op vooraf vastgelegde momenten. De bewerking kan het invoegen van een nieuw record of het aanpassen van een bestaand record zijn. De schrijver registreert op welk tijdstip deze taak is gestart, hoe en wanneer deze is beëindigd.

2.3. Stap 3: Testen van de systemen

Taak van de lezer De taak van een individuele lezer is om op vooraf vastgelegde momenten de data van de schrijver te 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 het start- en eindtijdstip 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 server. Met behulp van meerdere lezers kunnen er meer testpogingen voor het lezen van de data uitgevoerd worden door sommige gelijktijdig te laten plaats vinden. De verschillende starttijdstippen om de data te lezen worden verspreid tussen de verschillende lezers, op deze manier kan het exacte tijdstip dat een record correct gelezen worden exacter bepaald worden. Een voorbeeld van het plannen van 1 schrijver en 4 lezers wordt gegeven 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 door de queries van de lezers en schrijver. 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 onmiddellijke consistentie afdwingen een hogere belasting om de correcte waarde te lezen. Er zal dan twee of meer keren geprobeerd moeten worden.

Soorten uiteindelijke consistentie Met deze uitgevoerde testen en data kan aangetoond worden dat bepaalde systemen bepaalde uiteindelijke consistentie vereisen niet volgen. Het is mogelijk om met deze testen een tegenvoorbeeld te vinden

2. VOORGESTELDE TESTMETHODIEK

dat aantoon dat een systeem een eigenschap niet garandeert. Maar het vinden van geen tegen voorbeeld is geen bewijs van de eigenschap, maar kan wel een indicatie geven van de zeldzaamheid van de situatie.

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 voltooien van de schrijfactie*. Als dit voorkomt, gedraagt het systeem zich niet alsof er maar een enkele data-instantie is.

Read your own writes consistentie Deze garantie kan ontkracht worden indien een schrijver na het voltooien van zijn schrijfbewerking zijn eigen data opvraagt en niet de nieuwste waarde leest. Dit kan dezelfde connectie gebruik voor de schrijfbewerking zijn of een andere bewerking, een gebruiker moet dus meerdere connecties kunnen aanmaken.

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 door de schrijver verschillende schrijfacties na elkaar te laten uitvoeren. De lezer leest de records in een bepaalde volgorde. Indien de gebruiker de data van een latere schrijfactie leest maar nog niet van een schrijfbewerking die vroeger voltooid was, is deze eigenschap niet gegarandeerd. De eis kan strenger gemaakt worden door de schrijver tussendoor niet te laten lezen, dit zou andere resultaten kunnen hebben.

Monotonic Read consistentie In deze test leest de lezer continue hetzelfde record. Eenmaal deze een nieuwe versie heeft gelezen, mag deze nooit meer oudere data lezen. Indien dit wel het geval is, is ook deze eigenschap niet gegarandeerd.

Zoals beschreven 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 tegen voorbeeld te zijn voor een bepaalde consistentie categorie. Bij de beschikbaarheidstesten wordt er onderzocht hoe lang bepaalde bewerkingen onmogelijk zijn. Daarnaast wordt de gemiddelde vertraging tussen de normale situatie en de situatie met een service minder vergeleken voor de verschillende bewerkingen.

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 dan een tabel van cijfers. .

2.5 Conclusie

In dit hoofdstuk werden de verschillende stappen van de testmethode besproken. In het totaal bestaat de test uit 4 verschillende stappen. Deze testmethode biedt de mogelijkheid om op een analytische manier verschillende systemen gelijkaardig te testen naar hun gedrag in consistentie en beschikbaarheid. In het volgende hoofdstuk zal de implementatie besproken worden.

Hoofdstuk 3

Implementatie

In het vorige hoofdstuk is besproken wat de methodiek is voor het uitvoeren testen, maar hoe worden deze vereisten vertaald naar een werkende testprogramma? In dit hoofdstuk zal deze vertaling uitgelegd worden die in 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 de drie vermelde systemen in meer detail aan bod komen om zo de specifieke architectuur uit de doeken te doen. Tenslotte wordt besproken hoe de testsoftware geselecteerd en uitgebreid is. Dit is gevolgd door de configuratie van de testen voor de verschillende DBMS's: hoe de zijn de verschillende testen uitgevoerd en de resultaten verwerkt.

3.1 Selectie van de DBMS's

Bij de selectie van de systemen is er onderzocht of een systeem een bepaalde eigenschap al dan niet ondersteunt. In het totaal is de selectie op er vijf verschillende eigenschappen gebaseerd.

Vrije software Om de testen tussen verschillende DBMS's te vergelijken, is het nodig dat deze geïnstalleerd kunnen worden op de eigen infrastructuur. Daarnaast is er in deze thesis gefocust op systemen die ook nog 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 het opnieuw opstarten van een service dient er minder data over het netwerk gestuurd te worden aangezien dit van de harde schijf gelezen kan worden. Om deze reden hebben persistente systemen een voorkeur op deze die de data enkel in geheugen houden.

Replicatie Een deel van de benchmarking tool bestaat uit het onbeschikbaar maken van een service. Indien de data maar op een enkele server opgeslagen is, zal de data die de uitgeschakelde server opgeslagen was, voor geen enkele verbruiker

3. IMPLEMENTATIE

beschikbaar zijn. Met replicatie zal de data op verschillende servers opgeslagen worden, hierdoor kan de data beschikbaar gesteld worden met behulp van de andere servers.

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 ongewenst dat elke server al de data opslaat bij een grote dataset maar dat de data verspreid is over verschillende servers.

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. Het DBMS moet ondersteuning hebben 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, dit is niet hetzelfde als een range query waar de al de records opgevraagd worden tussen een begin en eind sleutel.

Alle systemen besproken in sectie A vervullen het eerste criterium. Een vergelijking voor de overige criteria is 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, in de thesis van mijn collega zijn dit Cassandra, Apache CouchDB, Riak en MySQL. Deze systemen zijn gekozen als voorbeeldsystemen die voldoen aan de criteria, maar ook andere DBMS's zouden gekozen kunnen worden.

3.2 Gedetailleerde bespreking van de model DBMS's

In dit gedeelte zal elk gekozen DBMS in meer detail uitgelegd worden. Een gemeenschappelijk element is dat elk systeem bestaat uit verschillende services en configuraties. Verschillende andere DBMS's bestaan uit slechts één service die intern de rest van de taken verdeeld, wat de installatie en configuratie kan vereenvoudigen.

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

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	Ja	Master-Master	Ja	Nee	Ja
	CoucheDB	Ja	Master-Slave	Ja	Ja	Ja
Key-Value	MongoDB	Ja	Master-Slave	Ja	Ja	Ja
	LightCloud (Tokyo)	Ja	Master-Master	Ja	Nee	Ja
Relationeel	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
Relationeel	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.

3.2.1 HBase

Data structuur[13]

De data in HBase is gestructureerd in tabellen, het aanmaken van een tabel gebuert door het definiëren van een schema. 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. s

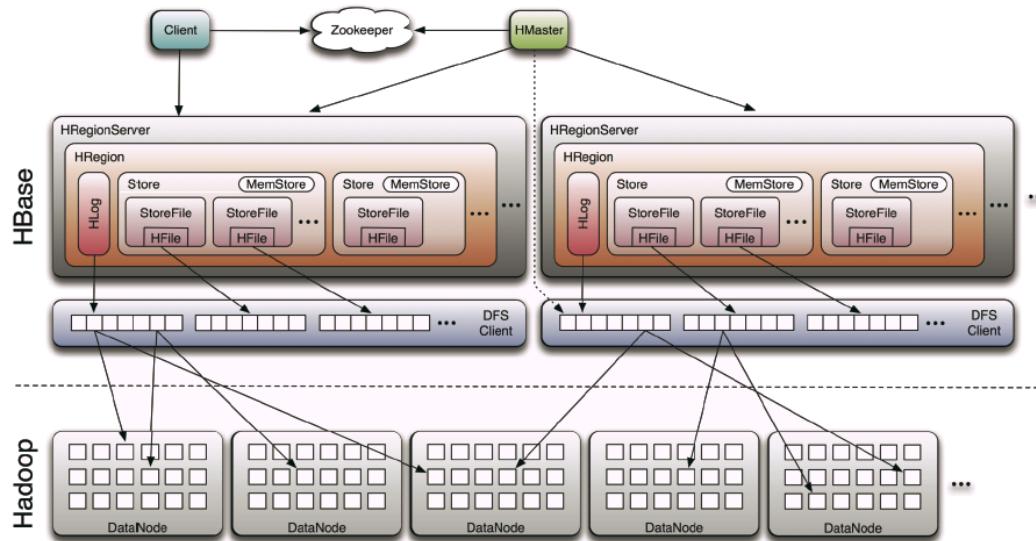
De operaties beschikbaar in dit systeem zijn: get (verkrijgen), put (invoegen), scan en delete (verwijderen). Het aanpassen van gegevens wordt uitgevoerd via een put waarbij een enkele kolomwaarde van een record kan aangepast worden. Een scan operatie heeft geen optie om het aantal op te halen records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

3. IMPLEMENTATIE

een enkele communicatie stap 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 respectievelijk Chubby[5] en Google File System[14]. Figuur 3.1 toont een overzicht van de architectuur van HBase. De drie systemen van HBase zullen kort besproken worden.



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 de master ook verantwoordelijk voor het toewijzen van dataverantwoordelijkheden. Zo is de master verantwoordelijke voor het opsplitsen van de data over verschillende regio's indien een tabel groeit en het toewijzen van een *HRegion* aan een *HRegionServer*.

Een *HRegionServer* is verantwoordelijke voor de data van een regio en voor het beheren van deze regio's. Een *HRegion* is een deel van een tabel die de data bevat, deze data wordt opgeslagen 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 één *namenode*, de master server, die de naamruimte en toegangscontrole onderhoudt, en *datanodes*. De data wordt opgedeeld in blokken die door een verzameling van

3.2. Gedetailleerde bespreking van de model DBMS's

datanodes wordt opgeslagen. Doordat niet elke node in deze verzameling zit, is er op deze manier datadistributie. Deze master/slave configuraties zijn verschillende services die de administrator afzonderlijk moet opzetten en configureren.

In de deze configuratie van HBase is HDFS de methode om data persistent op te slaan met automatische replicatie en datadistributie. 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 performant te zijn bij een dominantie van leesoperaties. HBase gebruikt Zookeeper voor het bijhouden van de status van HRegionserver, hun netwerklocatie en hun verantwoordelijkheden. De sessie wordt toegekend aan een HRegionServer die de verantwoordelijkheid krijgt voor een Region 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 deze lokale datanode zal elke regio bevatten van die de HRegionServer coördineert. 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. De verdeling en configuratie van de verschillende HRegion's wordt door het systeem zelf afgehandeld.

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 als 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

3. IMPLEMENTATIE

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 aan de bewerkingen. De bewerking kan voltooien nadat bijvoorbeeld de bewerking over het netwerk verstuurd of de primary heeft de data 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, kunnen er verschillende consistentie garantie verwacht worden. Een overzicht van al de 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.
Secondary	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 netwerk vertraging, ongeachte 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 totdat 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

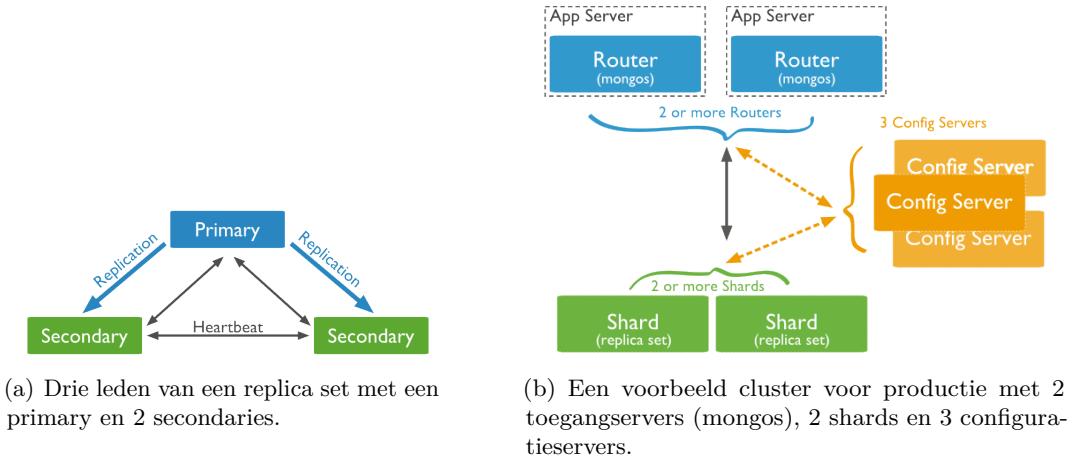
Tabel 3.2: MongoDB: Mogelijke configuraties bij lees- en schrijfbewerkingen

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 om de replicatievereisten in te vullen, daarna zal horizontale schaalbaarheid ondersteund worden.

Replicatie[26] Replicatie gebeurt door middel van een master/slave configuratie tussen verschillende *MongoD* instanties, of in MongoDB termen een primary/secondary. Een verzameling van deze MongoD instanties wordt een *replicaset* genoemd.

3.2. Gedetailleerde bespreking van de model DBMS's



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

Een replicaset kiest 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 cluster. In het geval van een zelfstandige instantie, zal de data niet gerepliceerd worden. Om deze reden wordt deze configuratie niet aangeraden voor productieomgeving. 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 aan de configuratie servers. Er kunnen een onbepaald aantal toegangsservers zijn in cluster.

3. IMPLEMENTATIE

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.

3.2.3 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 Pgpool-II hetzelfde principe als MongoDB.

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

Datastructuur

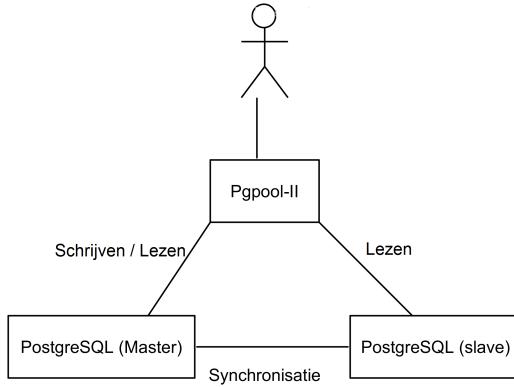
De data structuur en query mogelijkheden van Pgpool-II zijn gelijklopend aan deze van PostgreSQL. Net zoals in PostgreSQL bestaat het systeem uit een schema die verschillende databases kan bevatten. Een database bestaat uit een verzameling van tabellen, een tabel bevat de records. Voor het opslaan van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

Pgpool-II ondersteunt de SQL queries en vervult hiermee de volledige query benodigdheden 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.

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. 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 dataserver deze te laten uitvoeren zijn er een gelijke databases.



Figuur 3.3: Systeemarchitectuur van Pgpool-II.

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 welke PostgreSQL server dit zal uitvoeren. 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 net zoals integratie met Memcache ondersteund is.

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 licentie, hierdoor is de broncode vrij beschikbaar. YCSB is een uitgebreid benchmarking tool voor het uitvoeren van performantietesten, dit gebeurt door het meten van de gemiddelde vertraging op queries voor verschillende DBMS's voor een verschillend aantal queries per seconde. 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 voor Pgpool-II wordt rekening gehouden met het benodigd aantal records wat standaard in YCSB niet gebeurde.

De twee soorten testen, beschikbaarheids- en consistentietest, 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

3. IMPLEMENTATIE

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 zodat een conclusie kan getrokken worden over de beschikbaarheid van het volledige systeem.

Consistentietesten Voor de consistentie testen is er een extra module geïmplementeerd die het gedrag van sectie 2.3 uitvoert. In deze uitwerking leest de schrijver niet zijn eigen data, al zou dit eenvoudig mee geïmplementeerd kunnen worden. Dit is niet gemeten 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 benchmarking tool is beschikbaar op <https://github.com/thuys/YCSB-Implementation>.

3.4 Installatie en opstelling van de DBMS's en YCSB

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

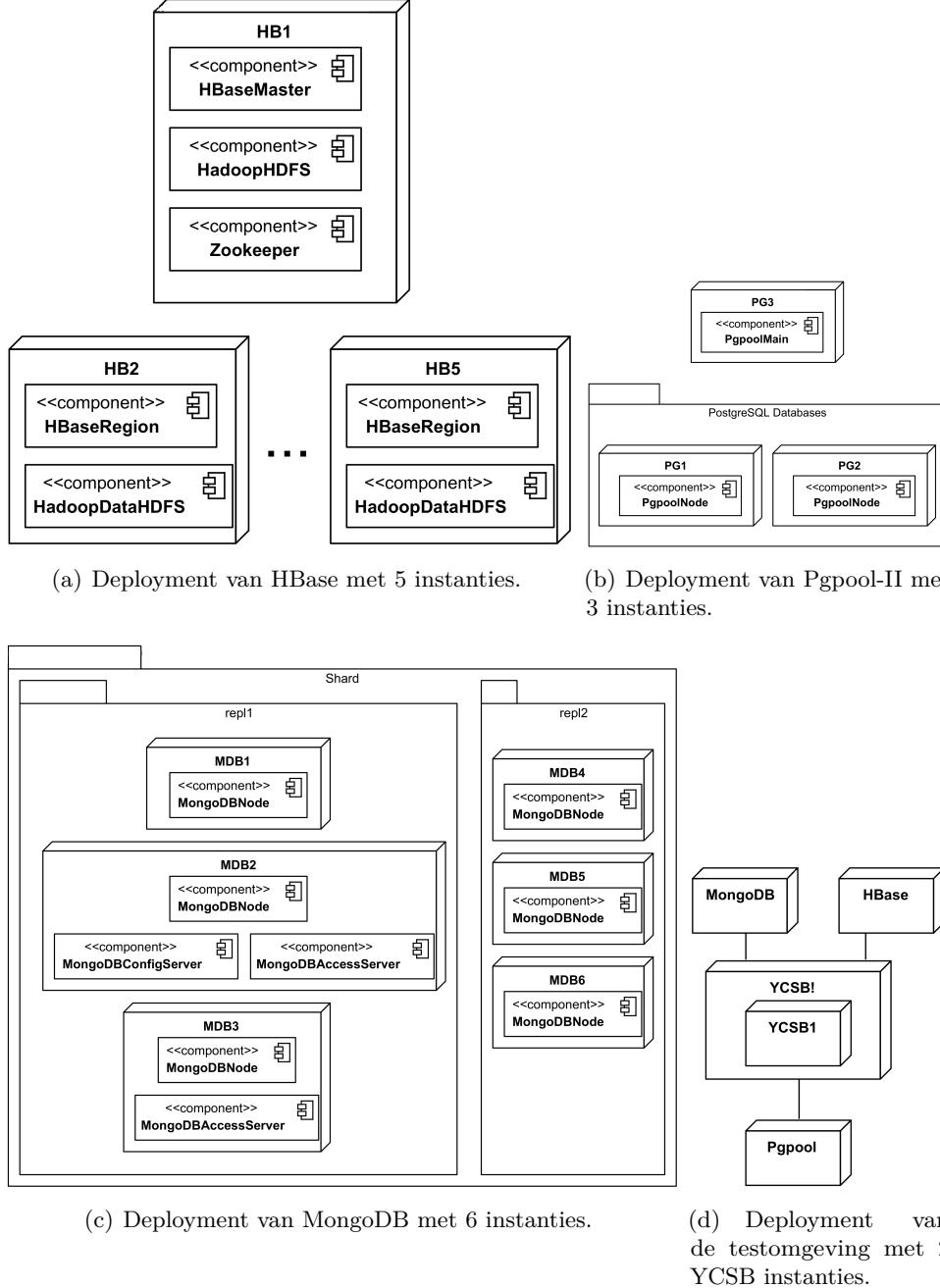
De automatisatie gebeurt met het Integrated configuration Management Platform (IMP) beschreven in [40]. Dit modular framework is uitgebreid met de drie DBMS's en YCSB waardoor de configuratie. Voor de installatie word de gewenste staat declaratief uitgedrukt, bij het uitrollen zal IMP deze staat op de verschillende systemen toepassen.

Een uitgebreider bespreking van de uitwerking in IMP kan gevonden worden in de IMP pakketten op GitHub, zie appendix C. Elk project bevat een bestand met een domeindiagram van het systeem, uitleg en voorbeeldcode.

Voor de uitvoering van de testen, is elk DBMS zo geconfigureerd dat het voor een minimaal aantal instantie heeft 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, routers en toegangsservers zijn minimaal opgezet omdat deze online blijven tijdens de testen.

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

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.



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

De testinfrastructuur is een IaaS (Infrastructure as a Service) gebaseerd op OpenStack¹.

¹<https://www.openstack.org/>

3. IMPLEMENTATIE

De infrastructuur bestaat uit 3 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 schijfruimte. De instanties worden willekeurig 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).

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 er altijd 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 op de eenvoudigste mogelijke manier uit te voeren. Een overzicht is getoond in figuur 3.4(d).

3.5 Uitvoeren van de kalibratie en testen

Voor het uitvoeren van de volledige benchmarking dient eerst de verdeling van de type queries gespecificeerd worden, deze zijn voor alle verschillende DBMS's 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.

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

Kalibratietesten Voor de kalibratie van de omgeving zijn er twee 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 de tweede test is het aantal gebruikers bepaalt door de uitkomst van de kalibratie 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	<i>zipfian (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: Kalibratie: Overzicht van de parameters voor het testen van het aantal gebruikers

Beschikbaarheidstesten Bij het uitvoeren van de testen op de beschikbaarheid van de verschillende systemen zijn de parameters vermeld 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 van de appendix. Voor Pgpool-II is er een extra commando toegevoegd dat na het herstarten van de systemen wordt uitgevoerd. Dit

3. IMPLEMENTATIE

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: Kalibratie: Overzicht van de parameters voor het testen van het aantal records per seconde

komt omdat er geen automatische recovery in Pgpool-II is. Tenslotte worden deze testen uitgevoerd op de datanodes, 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 records worden hier bewaard.

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

Consistentietesten Voor de consistentietesten 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. De aanname is dat er consistentie is zodra de gegevens beschikbaar zijn op al de verschillende instanties van een replicaset. Het testen van een cluster die twee replicasetten bevat, voegt enkel extra complexiteit toe. Deze test zou in de toekomst ook uitgevoerd kunnen worden op een cluster maar is in dit geval niet gedaan.

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

Overzicht van de testmethode 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.

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.

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 <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 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 met behulp van IMP zodat deze met slechts weinig kennis kunnen opgesteld worden. De testresultaten worden visueel voorgesteld zodat het eenvoudiger is om de data te verwerken.

Hoofdstuk 4

Observaties

Dit hoofdstuk behandelt de resultaten van de consistentie- en beschikbaarheidstesten uitgevoerd op HBase, MongoDB en Pgpool-II. In dit hoofdstuk zullen er geen verklaringen en analyses gemaakt worden over de testen, dit gebeurt in het volgende hoofdstuk.

De testen zullen zullen besproken worden in drie delen. Eerst zullen de resultaten van de kalibratie getoond worden met een selectie van het aantal gebruikers en bewerkingen per seconde, waarmee de beschikbaarheids- en consistentietesten uitgevoerd werden. 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 Kalibratie

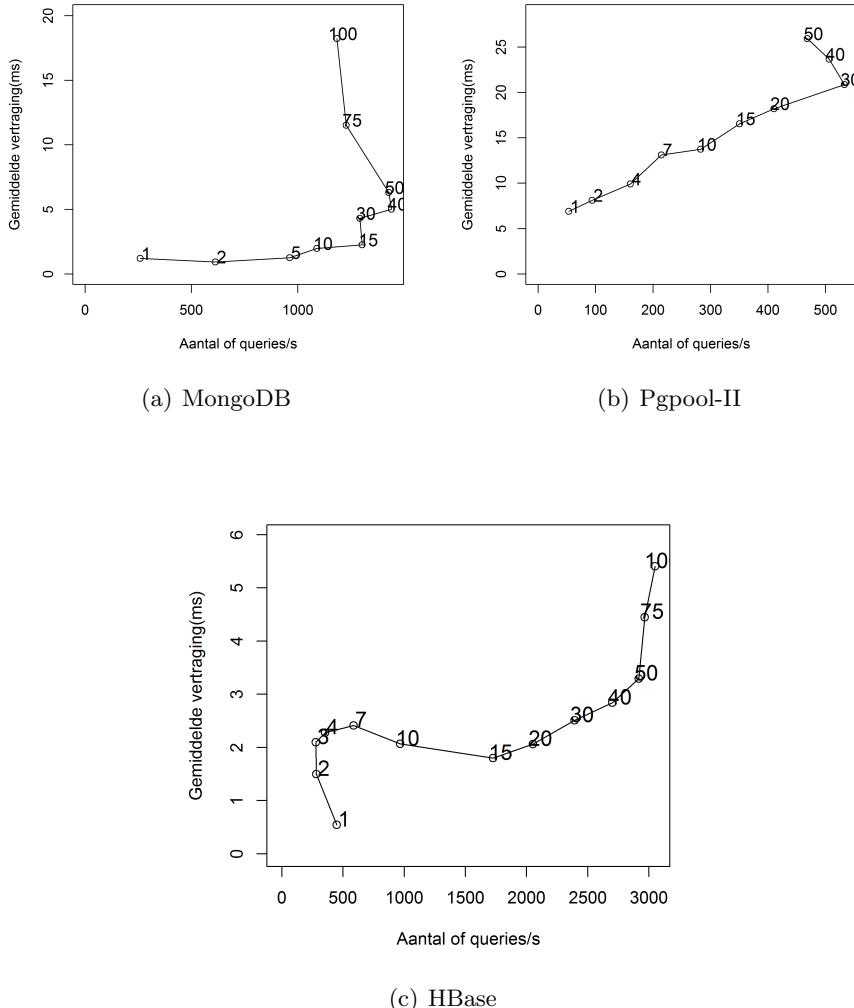
Aantal gebruikers De resultaten van de kalibratietest voor het aantal gebruikers kunnen gevonden worden in figuur 4.1.

Het aantal gebruikers wordt zo gekozen dat het totale aantal queries maximaal is of er 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: **Kalibratie:** Aantal gebruikers per test voor de verschillende DBMS's

4. OBSERVATIES



Figuur 4.1: **Kalibratie:** 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.

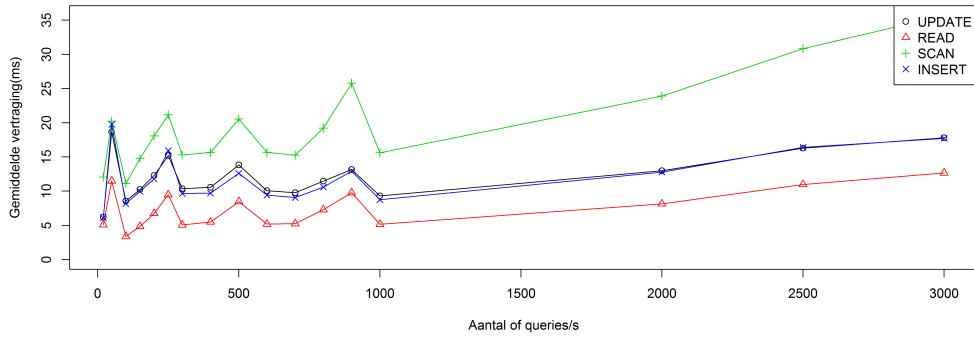
Op de x-as is het gemiddeld aantal queries per seconden getoond over een periode van 600s, op de y-as de gemiddelde vertraging in ms. Elk datapunt toont het totaal aantal queries per seconde en de gemiddelde vertraging voor verschillend aantal gebruikers (=connecties).

Aantal queries per seconde De resultaten voor de kalibratietest 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 een laag aantal queries. De onderste figuur toont op de y-as de verhouding tussen het

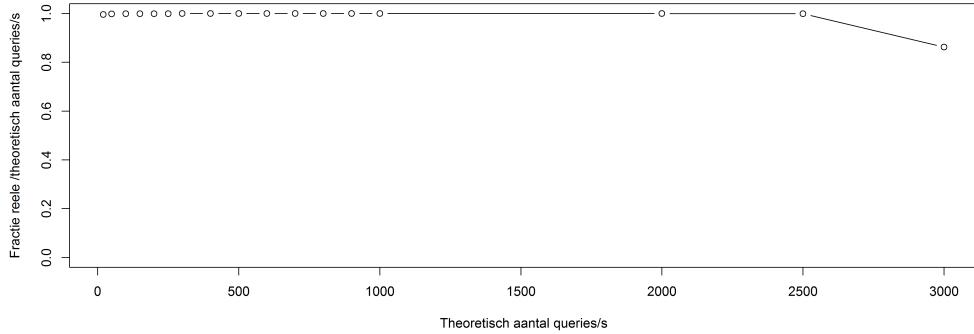
4.1. Kalibratie

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.

Voor elk DBMS kan met behulp van beide figuren een matige belasting gekozen worden. 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. Het benaderen van de waarde 1 bij de tweede figuur is nodig zodat het theoretisch aantal bewerkingen ook in de praktijk gehaald worden, anders is er ergens een bottleneck en is de belasting dus niet matig. De testen zouden ook uitgevoerd kunnen worden onder een andere belasting en de resultaten zouden vergeleken kunnen worden.



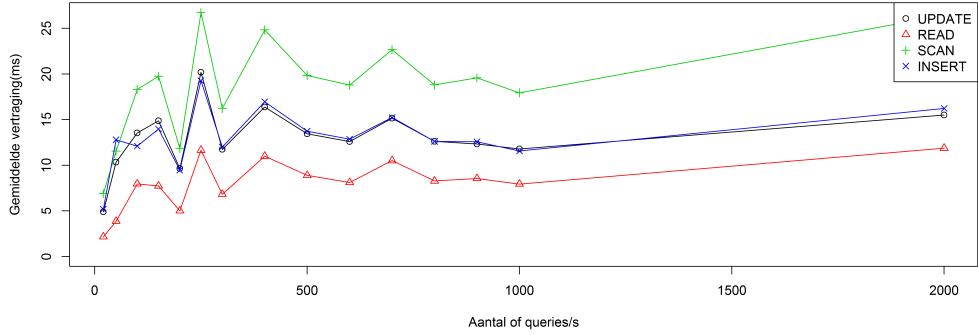
(a) Gemiddelde vertraging bij een verandering in aantal queries per seconde.



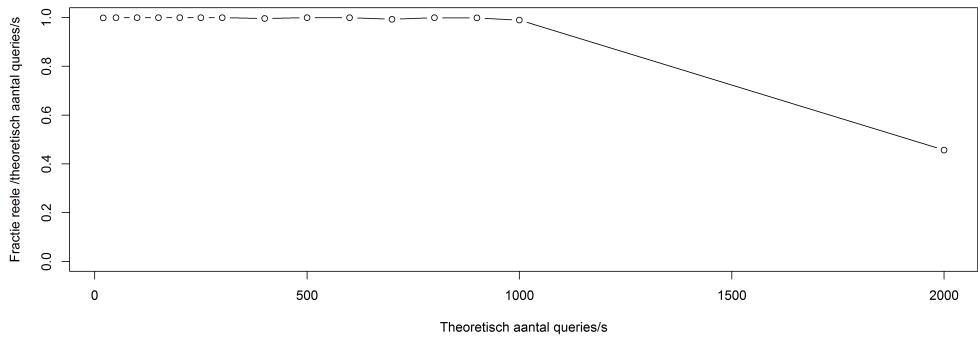
(b) Verhouding van het aantal uitgevoerde queries ten opzichte van het aantal gevraagde queries.

Figuur 4.2: **Kalibratie:** 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.

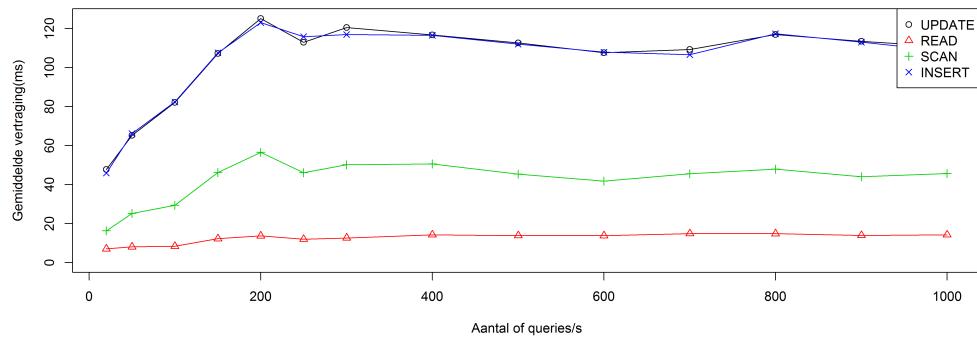


(b) Verhouding van het aantal uitgevoerde queries ten opzichte van het aantal gevraagde queries.

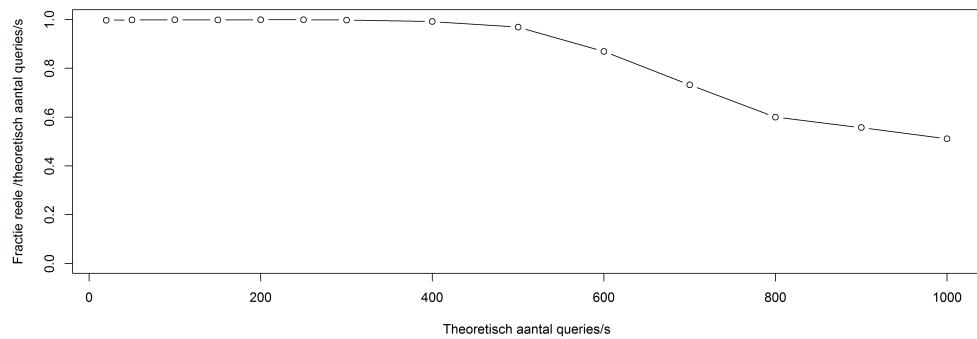
Figuur 4.3: **Kalibratie:** Overzicht van de vertraging t.o.v. het theoretisch aantal aanvragen met een vergelijking hoeveel werkelijke aanvragen er waren voor MongoDB.

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

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



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

4. OBSERVATIES

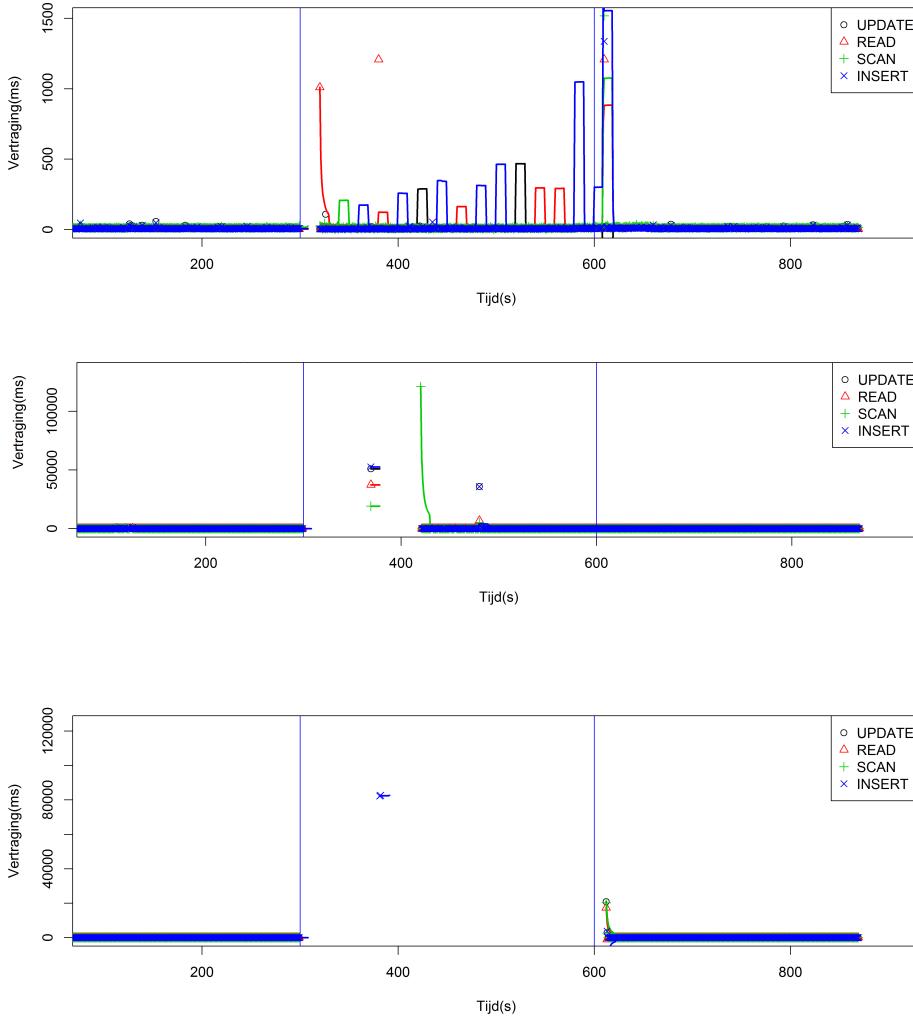
4.2 Beschikbaarheidstest

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 via 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



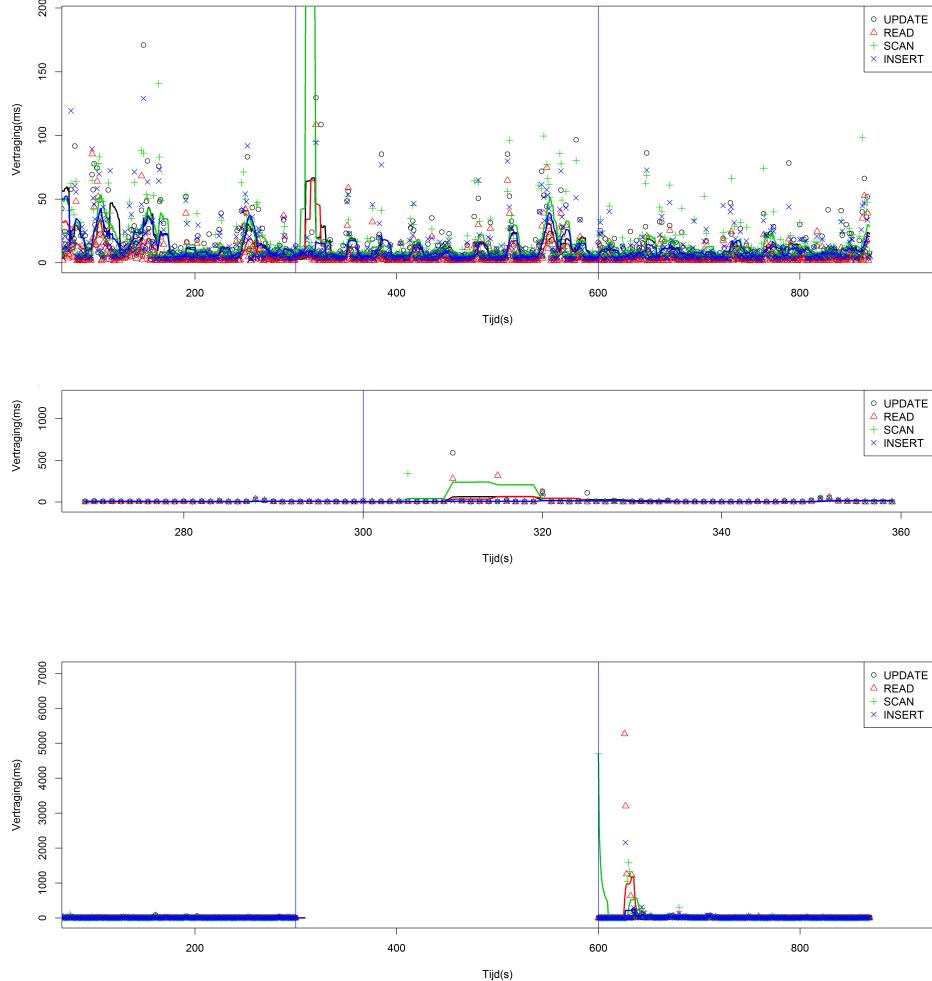
Figuur 4.5: **Beschikbaarheid van HBase**

Bij HBase zijn er verschillende reacties op het stopzetten van een node. Zo kun je in de eerste figuur een zachte stop zien. Bij het zacht stoppen van een instantie, is er een onderbreking van ongeveer 20 seconde. Daarna kunnen er af en toe nog verhogingen in de queries optreden. De tweede figuur stelt een netwerkonderbreking voor, er is een onderbreking van gemiddeld ongeveer 100 seconden. Daarna is het terug stabiel.

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. De laatste figuur toont het resultaten wanneer er geen queries mogelijk zijn.

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

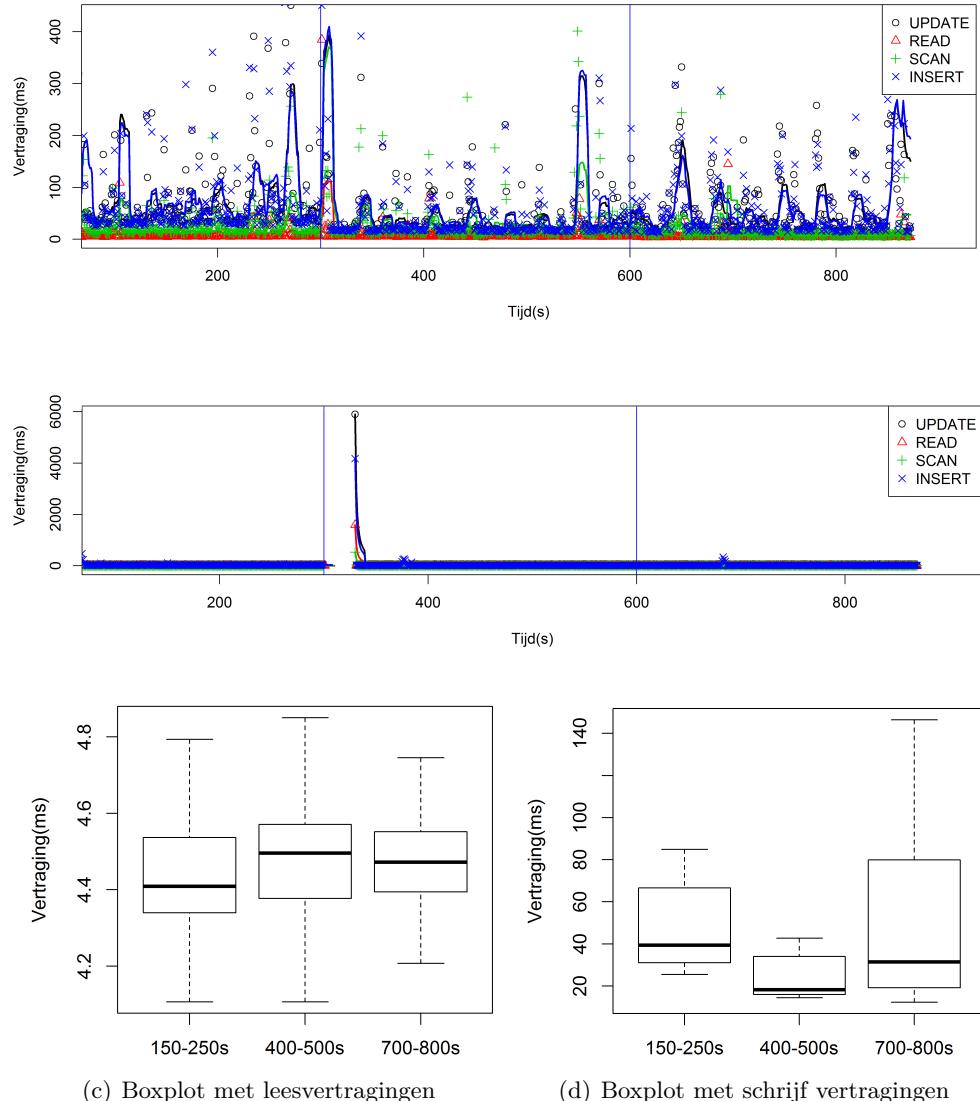


Figuur 4.6: Beschikbaarheid van MongoDB

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 illustreert een zoom naar de stop.

Bij het onderbreken van het netwerk is er in bepaalde gevallen geen significante verandering, op andere momenten is een gedrag soortgelijk aan dat bij een zachte stop op 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**

Bij Pgpool-II is er geen verschil tussen een harde of zachte 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 is de onderbreking voorbij. Een voorbeeld bevindt zich in de middelste figuur.

Tijdens de onderbreking is er een verandering van de vertraging van de schrijfbewerkingen: 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 nadat deze opnieuw online gebracht hebben, lukt niet tijdens de testen. Enkel als alle connecties verbroken zijn, wat het geval is na de testen, slaagt het herstel.

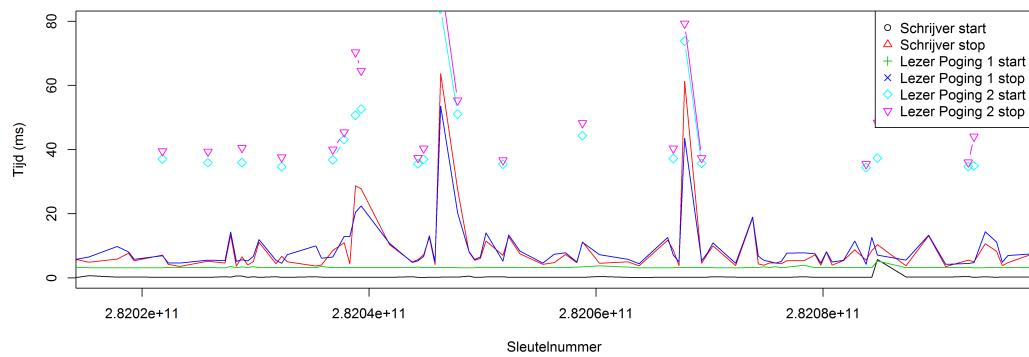
4. OBSERVATIES

4.3 Consistentietest

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

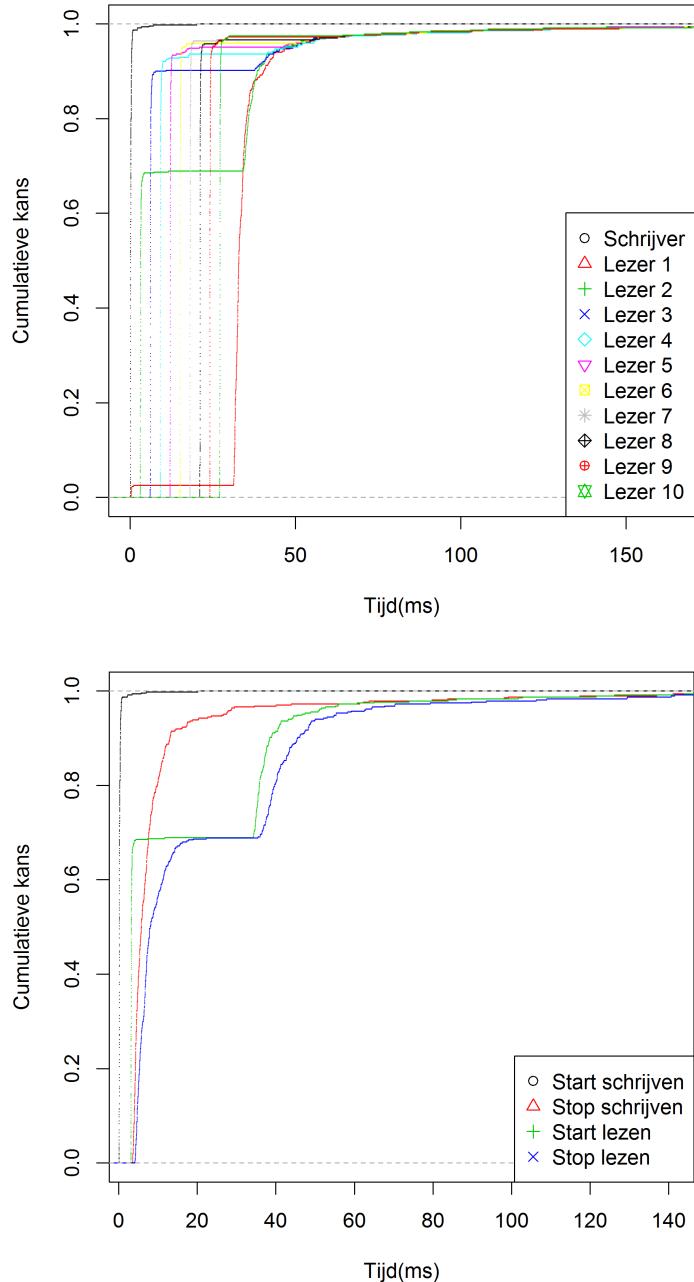
Voor de consistentie testen wordt er op de x-as de start- en/of stoptijdstippen van de verschillende soorten getoond. Het verschil tussen de y-waarde van de start- en stoptijdstippen geeft aan hoeveel queries er op dat moment uitgevoerd worden. De getoonde tijdstippen van een lezer zijn de eerste keer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

De tweede soort figuren heeft op de x-as het sleutel nummer en op de y-as wordt de vertraging getoond. De grafieken die geplot worden zijn de het start- en eindtijdstip van zowel de schrijfbewerking als de verschillende pogingen van een bepaalde lezer.



Figuur 4.8: **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.

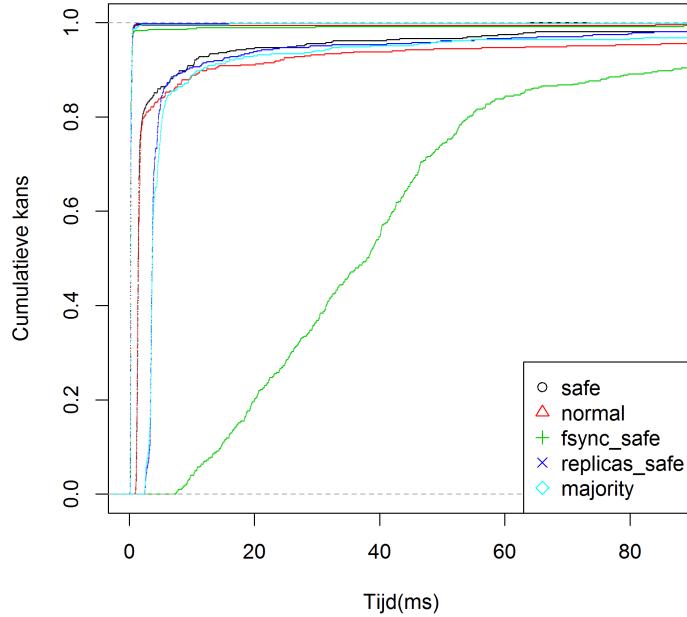


Figuur 4.9: Consistentie van HBase

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.

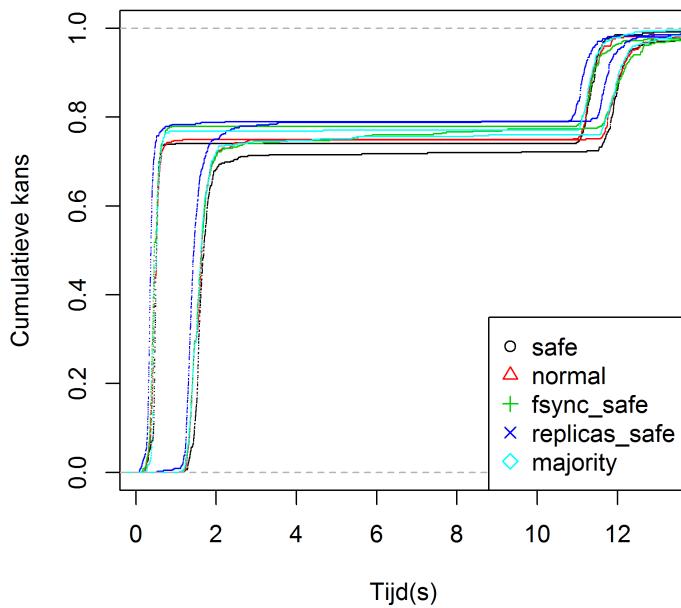
Figuur (a) toont een overzicht van de verschillende starttijdstippen voor het lezen van consistentie data. Figuur (b) toont de start- en eindtijdstippen voor lezer 2 naast deze voor de schrijver. Lezer 2 start met op 3ms en leest vervolgens elke 30ms tot de nieuwe waarde is gelezen. De maximale waarde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

4. OBSERVATIES

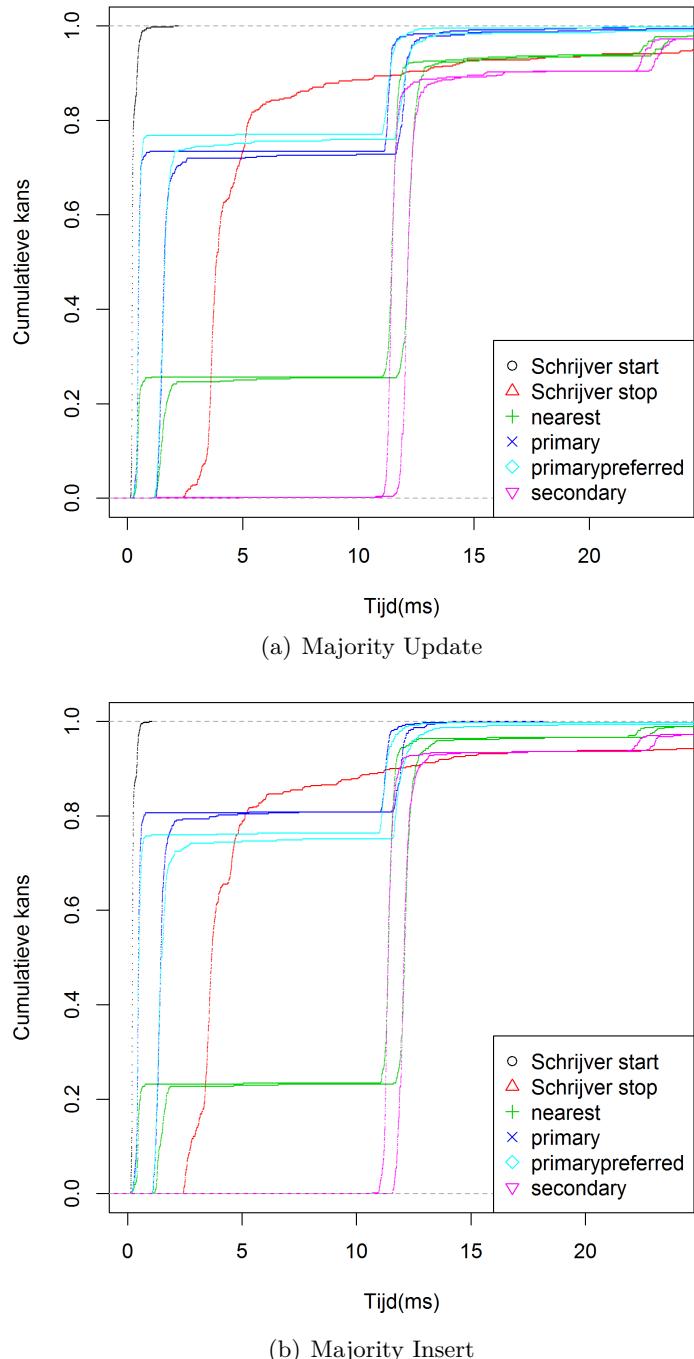


Figuur 4.10: **Consistentie van MongoDB:** Verloop van schrijfoperaties

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



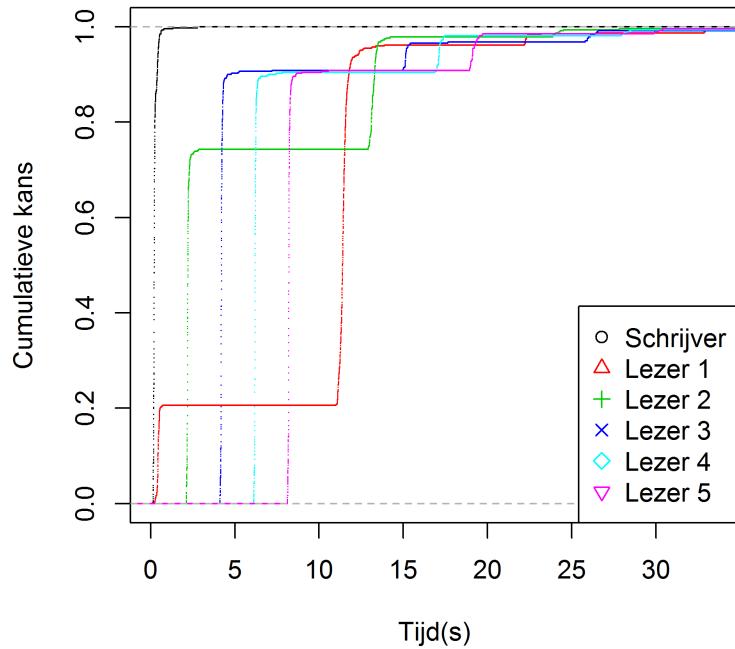
Figuur 4.11: **Consistentie van MongoDB:** Leesgedrag bij verschillende schrijfconfiguraties
Een vergelijking van het leesgedrag onder verschillende schrijfconfiguratie toont dat er geen significant verschil is. In de figuur is een 99-percentiel getoond voor een update operatie met als leesactie primary, de start- en stoptijden zijn in dezelfde kleur.



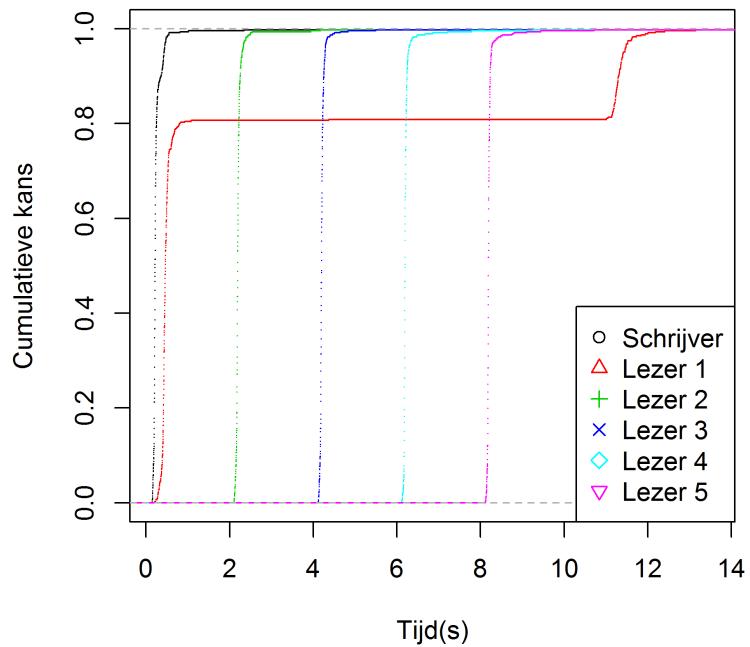
Figuur 4.12: **Consistentie van MongoDB:** Insert vs Update voor lezer 2

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.

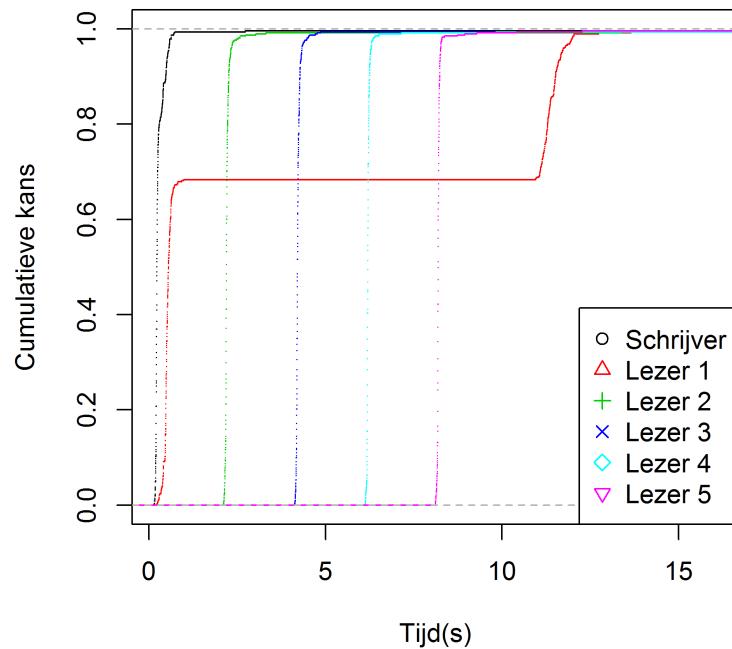
4. OBSERVATIES



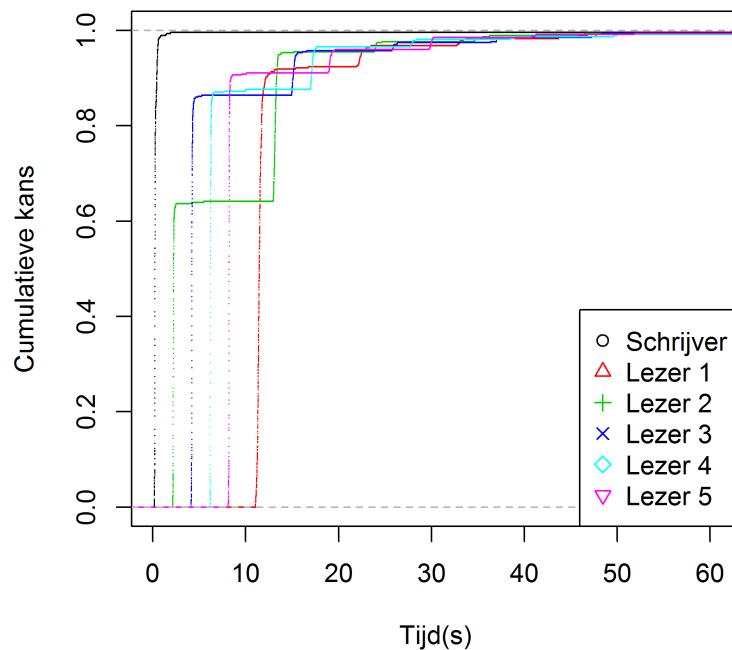
Figuur 4.13: **Consistentie van MongoDB:** Overzicht van startmoment van consistente leesactie voor het lezen op de nearest



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

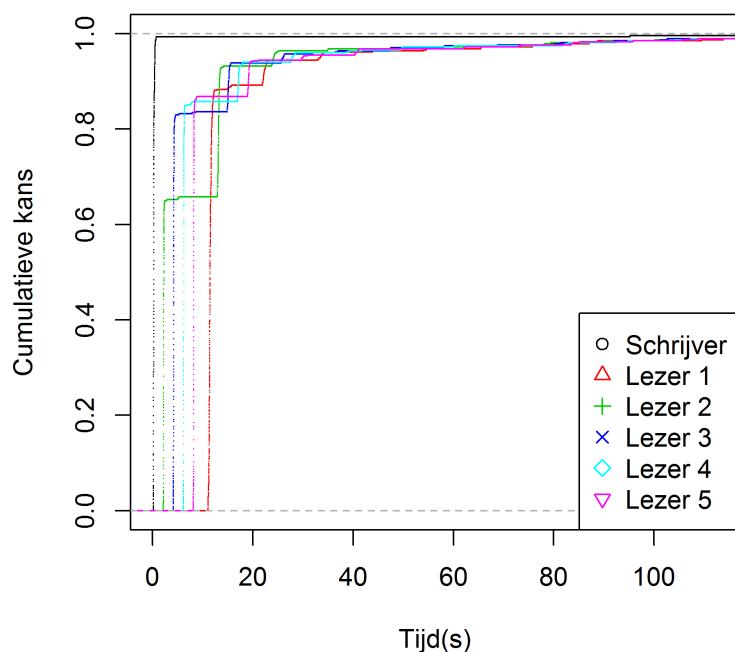


Figuur 4.15: **Consistentie van MongoDB:** Overzicht van startmoment van consistente leesactie voor het lezen op de primarypreferred



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

4. OBSERVATIES



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 voor de drie systemen naar beschikbaarheid toe. Ook in consistentie zijn er verschillen tussen HBase en MongoDB. 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 kalibratie 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 Kalibratie

Over de kalibratie 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 vertragingen in MongoDB zal dieper ingegaan worden. Bij het toenemen van het aantal gelijktijdige gebruikers neemt niet enkel de gemiddelde vertraging toe, maar ook de standaard deviatie groeit significant. Sommige bewerkingen verlopen snel, andere bewerkingen staan zeer lang in de wachtrij. De reden hiervoor is een lezers/schrijvers locking systeem op de 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.

Het gevolg van deze werkwijze is dat indien veel gebruikers actief zijn, de duur van een bewerking moeilijk te voorspellen is. Voor de resultaten van de benchmarking tool is het best om zo weinig mogelijk variatie binnen een testrun te hebben, vandaar is er gekozen voor weinig gelijktijdige gebruikers.

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 drie testen in detail besproken worden, daarna zal een vergelijking tussen de drie systemen besproken worden.

HBase Bij HBase heeft een bepaalde HRegionServer 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 door de HRegionServer of er moet gewacht worden tot deze verlopen is, enkel op die momenten kan er een nieuwe HRegionServer 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 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).

Het stopzetten van een HRegionServer is enkel merkbaar als er data gelezen of geschreven wordt waarvoor deze server verantwoordelijk is. In de testen worden vele bewerkingen per seconde uitgevoerd per connectie. De kans dat de HRegionServer gecontacteerd wordt kort na het stopzetten is daarom zeer groot.

Zodra er een herverdeling is van de HRegions over de aanwezige HRegionServers, verdwijnt deze verhoogde vertraging. De onderbreking neemt enkele seconden in beslag.

Netwerk onderbreking Bij een netwerk onderbreking, worden de queries tijdelijk stopgezet en falen de queries in tussentijd. Deze onderbreking duurt significant langer, tot 180 seconden, 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 deze permanente onderbreking af en toe ontstaat, is niet gevonden.

Herstel van de instantie Het herstel van de server zal automatisch op een asynchrone manier gebeuren. Er valt te configureren hoeveel netwerkverkeer er maximaal per seconde zal worden gebruikt voor de synchronisatie. Het herstel is niet merkbaar voor de gebruiker op de vertraging van de bewerkingen.

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 als laatste contact met de primary heeft gehad zal zijn rol overnemen indien de primary onbereikbaar wordt[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 een replicaset bestaat uit drie server in onze configuratie. In de standaard modus wordt er enkel gelezen naar en geschreven van de primary, indien deze onbereikbaar wordt is er een onderbreking. Na het kiezen 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 beschikbaar 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.

5. ANALYSE VAN DE RESULTATEN

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 worden alle bewerkingen geblokkeerd tot een antwoord of de time-out van de connecties naar de dataserver die standaard 30 seconde is. In dit geval is er een time-out en gebeurt de interne overschakeling. Hierbij worden alle connecties van de gebruikers verbroken en de onbereikbare server niet meer gebruikt. Daarna kan de gebruiker opnieuw verbinden met Pgpool-II en queries uitvoeren.

Vermindering van schrijfvertraging De reden tot de vermindering van de schrijfvertraging is te verklaren door de wijze waarop Pgpool-II de schrijfbewerkingen uitvoert. Deze zullen eerst op de master uitgevoerd worden en vervolgens op de verschillende slaves. Bij het wegvalLEN van een instantie is er in de testopstelling nog maar een enkele dataserver over, hierdoor duurt een schrijfactie maar half zo lang. De leesacties duren ongeveer even lang aangezien er nog steeds gelezen wordt van een dataserver.

Herstel van de instantie Bij het opnieuw inschakelen van een instantie dient in Pgpool-II het herstel handmatig opgestart 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 dit 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 was er continue dataverkeer en werden er nieuwe databasebewerkingen naar Pgpool-II gestuurd, hierdoor slaagde het herstel niet.

Samenvatting Zowel HBase als MongoDB volgen de status de hele tijd op. Bij HBase is dit de verantwoordelijkheid van Zookeeper waarbij onder andere de sessieduur geconfigureerd worden. Bij MongoDB wordt dit afgehandeld door de verschillende services zelf en is er geen configuratiemogelijkheid. Pgpool-II gebruikt een principe 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.

Doordat er een matige belasting is van de verschillende systemen is het effect tijdens het onbeschikbaar zijn van de service beperkt naar de duur van de bewerkingen

als ook het herstel. De verschillende werkende servers kunnen de extra belasting opvangen, bij een hoge belasting zou het kunnen zijn dat er wel een effect is.

	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

	Automatisch herstel
HBase	Ja
MongoDB	Ja
Pgpool-II	Nee

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

Implicatie van de resultaten voor een gebruiker De drie systemen kunnen opgesplitst worden in twee groepen. Indien de ontwikkelaar van een softwaresysteem een systeem wilt dat keer op keer hetzelfde reageert op het onbeschikbaar worden van een server, dan kiest hij het best voor Pgpool-II. Het nadeel van deze keuze is dat voor het herstel tijdelijk de database connecties moet verbroken worden. Ook Pgpool node kan een limiterende factor worden naar de schaling, maar met de parallelle mode van Pgpool-II kan dit overwonnen worden.

Wil men daartegen een systeem met automatisch herstel, dan kan men kiezen voor HBase of MongoDB. Als de ontwikkelaar het gedrag bij faling in detail zelf wilt configureren, heeft HBase vele configuratie mogelijkheden. MongoDB is hierin meer beperkt maar biedt een systeem aan dat standaard sneller herstelt. Voor beide systemen kan het soms nodig zijn om een connectie te verbreken en een nieuwe connectie op te zetten als een service onbeschikbaar wordt.

5.3 Consistentietest

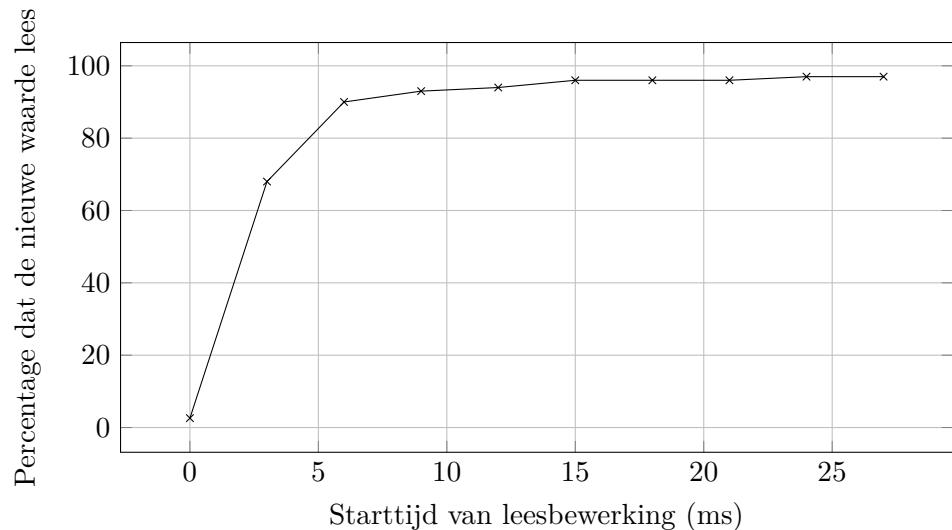
HBase HBase garandeert strikte consistentie op een enkel record en hoe deze garantie tot uitvoering wordt gebracht, is zichtbaar in figuren 4.8 en 4.9(b). Een leesbewerking wordt namelijk op wacht gezet tot de schrijfbewerking voltooid is. In de eerste figuur wacht de leesbewerking met afronden van zijn bewerking tot de schrijfbewerking is voltooid. Indien een leesbewerking vroeger gedaan heeft, moet deze een tweede poging ondernemen omdat deze de eerste keer oude data heeft gelezen. In de tweede figuur heeft de lijn van het stoppen met lezen van de correcte waarde continue een hogere waarde dan het voltooien van de schrijfbewerking.

Dit gedrag is te verklaren doordat HBase gebruik maakt van lezer/schrijvers locking,

5. ANALYSE VAN DE RESULTATEN

hierdoor moet een leesactie wachten tot een schrijfactie de lock terug vrij geeft. In figuur 5.2 wordt het lees- en schrijfmodel in detail uitgelegd aan de hand van de beschrijving 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 langer duren door het wachten op de lock.

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 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 grafische voorstelling voor de vier leesconfiguratie bevindt zich in figuur 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 schrijf-

Schrijven

1. Lock de rij(en), om te beschermen tegen gelijktijdige lees- en schrijfactiviteiten.
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-Value 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]

configuraties geven geen garanties tijdens het uitvoeren maar geven verschillende garanties als de schrijfactie voltooid is.

Uit tabel 5.3 blijkt dat het in MongoDB niet gegarandeerd is dat als een lezer de nieuwe waarde leest, al de andere lezers dit ook zullen doen. In het voorbeeld was het schrijven nog niet voltooid maar toch las een bepaalde lezer de nieuwe data al. Een leesbewerking die later gestart is, leest de oude waarde nog. Dit kan verklaard worden doordat het verschillende servers zijn waarop gelezen wordt. Het waren verschillende lezers, maar de MongoDB driver controleert periodiek welke server het dichtste bij is en kan een nieuwe server kiezen als nearest. Een hypothese is dat de leesserver kan veranderen tussen twee bewerkingen gebeuren en hierdoor na het lezen van de nieuwe waarde op bijvoorbeeld de primary, de tweede keer nog de oude data op een secondary gelezen wordt. Om deze reden wordt er verondersteld dat er geen garantie is op monotone consistentie als men niet lees van de primary.

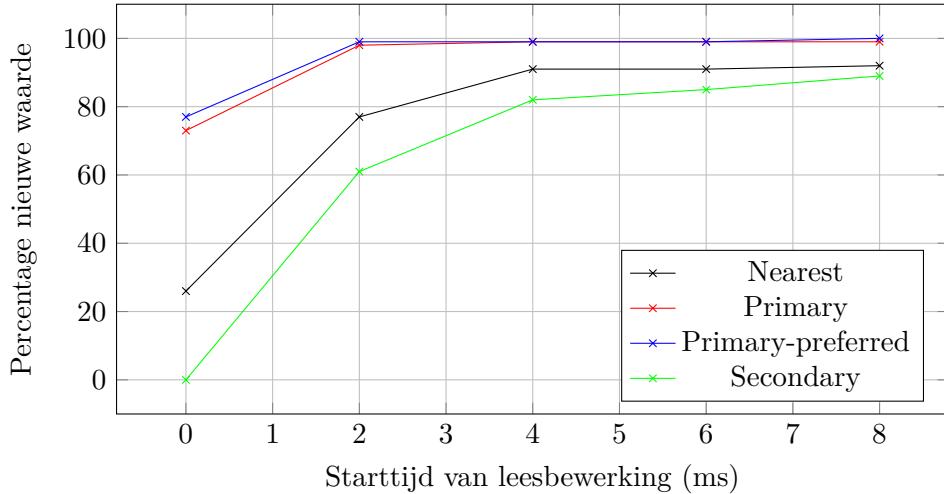
In de testomgeving is er een uniforme netwerkvertraging tussen alle servers. De theoretische kans dat een dichterbijzijnde server een primary is, is 1 op 3, voor een secondary is dit 2 op 3. Het gedrag in de praktijk voor consistentie heeft minder dan 5% afwijking van dit theoretisch model. De testresultaten van nearest kunnen dus ook afgeleid worden in de toekomst uit de resultaten van primary en secondary.

Tenslotte hebben primary en primary-preferred in deze testen geen significante verschillen. Dit komt omdat de primary heel de tijd beschikbaar is en er niet overgeschakeld moet worden naar de secondary.

5. ANALYSE VAN DE RESULTATEN

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



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.

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

Een hypothese is dat het falen van de primary de consistentie garanties zou beïnvloeden. Indien de laatste schrijffactie nog niet gerepliceerd is nog een secondary maar al wel gelezen is door een gebruiker, zal de data niet beschikbaar zijn op de nieuwe primary. De nieuwe data zal voor lezers dus niet meer beschikbaar zijn, er zou dus geen strikte consistentie in de primary leesconfiguratie kunnen zijn. Dit gedrag is

wel niet bevestigd met testen in de praktijk. Een andere hypothese is dat HBase heeft deze situatie niet voor omdat de leesbewerkingen uitgesteld worden tot na het voltooien van de schrijfacties.

Implicatie van de resultaten voor een gebruiker MongoDB en HBase bieden in de standaardconfiguratie ongeveer dezelfde garanties. Indien het voor de ontwikkelaar belangrijk is dat nieuwe data zo snel mogelijk beschikbaar is of zelf de consistentie-eigenschappen wenst te kiezen, dan is MongoDB een goede keuze. Mag de data slechts beschikbaar zijn nadat de schrijfbewerking voltooid is, dan kan er voor HBase gekozen worden. Een situatie die deze laatste garantie kan afdwingen, is bijvoorbeeld een beurssysteem: het is mogelijk voor andere gebruikers om een aankoop te weten te komen vooraleer de koper zelf de bevestiging heeft gekregen.

5.4 Conclusie

De drie systemen hebben verschillende aanpak voor beschikbaarheid en consistentie. Pgpool-II heeft door zijn centrale aanpak met behulp van de Pgpool-II node een consequent gedrag bij het onbeschikbaar worden van een dataserver. Daartegenover staat wel dat dit systeem geen automatisch herstel heeft maar manuele interventie vereist dat het volledige systeem tijdelijk onbereikbaar maakt.

Het gedrag van MongoDB bij het onbeschikbaar worden van een instantie, is weinig configurerbaar. De resultaten zijn voorspelbaar maar het is mogelijk dat een database connectie niet automatisch correct wordt hersteld. Nieuwe data in veel gevallen al beschikbaar voor het voltooien van de schrijfbewerking. Daarnaast zijn er veel mogelijkheden om de garanties te configureren voor het lezen en schrijven van data. Indien er gelezen wordt van de primary bevestigen de testen dat er strikte consistentie is, het is wel onduidelijk welke garanties er zijn bij het falen van een primary.

Met behulp van Zookeeper kan de reactie op het onbeschikbaar zijn van een server in HBase geconfigureerd worden, onder andere door de duur van een sessie te veranderen. Ook hier is het mogelijk dat een database connectie niet automatisch wordt hersteld. Het systeem garandeert strikte consistentie. Een leesopdracht van een record die binnenkomt tijdens het schrijven of aanpassen van dat record, zal in de wachtrij gezet worden om de nieuwe data te lezen nadat de schrijfbewerking voltooid is. In de testen werd de nieuwe data nooit gelezen voor het voltooien van de schrijfbewerking.

Hoofdstuk 6

Conclusie

Deze thesis heeft de kloof verkleint tussen de informatie die nodig en beschikbaar is voor het selecteren van een DBMS. Dit is gebeurd door het ontwikkelen een nieuwe benchmarking tool om consistentie- en beschikbaarheidseigenschappen te onderzoeken. Daarnaast werd deze tool toegepast om drie verschillende voorbeeldsystemen te onderzoeken en vergelijken. Al deze software is beschikbaar en eenvoudig bruikbaar voor andere gebruikers door de hulp een automatische installatie en configuratie van de benchmarking tool en de DBMS's.

In dit hoofdstuk zullen eerst de uitdagingen besproken worden die overwonnen werden bij deze thesis. Dit wordt gevolgd met een overzicht voor potentieel verder onderzoek. Tenslotte wordt de thesis afgesloten met een samenvatting van de belangrijkste bijdragen.

6.1 Uitdagingen

Handmatig opstellen van de DBMS's De grootste uitdaging van de thesis was om de drie verschillende DBMS's handmatig te installeren en configureren tot een feilloos werkend systeem. Dit was een uitdaging omwille van verschillende redenen.

Allereerst bestaan alle drie de systemen uit veelvoud van services en verschillende services gebruiken een andere configuratie methode. Bij HBase zijn er niet minder dan 5 verschillende services die elk geconfigureerd worden door middel van bestanden. Bij MongoDB verloopt de configuratie op zijn beurt via de terminal.

Daarnaast zijn deze systemen nog volop in ontwikkeling, hierdoor is de documentatie vaak niet al te uitgebreid en zijn de foutmeldingen vaag. Dit was onder andere het geval bij Pgpool-II waarbij er enkel documentatie was voor Debian, maar de testinfrastructuur werkt met Fedora. Onder andere het gebruik van de beveiligingsmodule SELinux heeft de correcte werking van het herstel van Pgpool-II lange tijd gehinderd.

6. CONCLUSIE

Automatische installatie en configuratie van DBMS's Als een systeem manueel kan opgezet worden, betekent dit niet dat er een onmiddellijke vertaling is naar het automatisch opstellen met behulp van IMP, hiervoor zijn er twee redenen.

Allereerst kan het zijn dat een lijn in de terminal niet hetzelfde effect heeft dan wanneer deze in een bashscript wordt uitgevoerd. Het commando kan afhankelijk zijn van de huidige situatie van het DBMS. Bij MongoDB was het nodig om eerst de huidige configuratie te weten van de cluster voor het toevoegen of verwijderen van servers of replicaset's.

Daarnaast kon IMP op het moment van de ontwikkeling nog geen afhankelijkheden aan tussen verschillende servers. Onder andere in MongoDB zorgde dit voor uitdagingen. Een replicaset kan bijvoorbeeld maar geïnitialiseerd worden wanneer alle servers online zijn. Met behulp van de thesis van Harm De Weirdt[10] is dit nu wel mogelijk in IMP. Als zijn methode toegepast wordt, zou de uitrol eenvoudiger kunnen verlopen.

Hoewel de automatisatie de nodige tijd en moeite heeft gekost, heeft dit zijn plaats in deze thesis. Allereerst is het voor andere gebruikers eenvoudiger om de systemen op te zetten en de testen uit te voeren, dit was ook het geval voor mijzelf. Tijdens de ontwikkeling van de benchmarking tool was het af en toe nodig om al de servers te resetten. Het opnieuw opzetten van de infrastructuur ging veel vlotter, ook kon er eenvoudiger geëxperimenteerd met verschillende configuraties.

Uitvoeren van de benchmarking tool Een laatste uitdaging was het ontwikkelen van de benchmarking tool en de configuratie van de testen .De uitdaging hierbij zat hem in de tijd die nodig was om de testen op punt te stellen. Over de configuratieparameters is verschillende keren geïtereerd en elke iteratie kost veel tijd. Deze iteraties waren nodig bij het veranderen van de infrastructuur of de configuratie. Beiden kunnen als gevolg hebben dat het nieuwe systeem sneller of trager is. In de uiteindelijke configuratie duurt het uitvoeren van een enkele kalibratietest ongeveer 15 minuten, bij een consistentietest is dit ongeveer 10 minuten, bij een beschikbaarheidstest loopt dit op tot 20 minuten.

6.2 Verder werk

In deze thesistekst zijn de eerste resultaten en conclusies naar beschikbaarheid en consistentie getrokken voor drie verschillende DBMS's. Een uitbreiding van het aantal geteste systemen zorgt er allereerst voor dat meer vergelijkend materiaal is maar daarnaast kunnen ook categorieën gevormd worden zoals bij het datamodel.

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 een ping tijd rond de 0.4ms, maar wat is bijvoorbeeld de invloed van deze parameter in de testen, hetzelfde geldt voor het

6.3. Evaluatie van de doelstellingen en bijdragen

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 het fysiek scheiden van de lezers en schrijven bij de consistentietest, wat een mogelijke invloed kan hebben op de resultaten. De beschikbaarheidstesten kunnen ook getest worden met verschillende fysieke gebruikers en onderzocht worden of deze hetzelfde gedrag hebben.

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?

6.3 Evaluatie van de doelstellingen en bijdragen

Deze thesis heeft met behulp van drie doelstellingen de kloof tussen de benodigde en beschikbare informatie kleiner gemaakt in relatie met de consistentie en beschikbaarheid van een DBMS. De verschillende doelstellingen en hun bijdrage zullen in deze sectie overlopen worden.

Ontwikkelen van een benchmarking tool In hoofdstuk 2 is de algemene testmethode beschreven om de beschikbaarheid en consistentie van DBMS's te onderzoeken. Dit gebeurt door middel van verschillende stappen die bestaan uit het opstellen van de infrastructuur, uitvoeren van de testen en het analyseren van de resultaten. Hoofdstuk 3 bespreekt hoe de verschillende stappen in de praktijk worden geïmplementeerd. De benchmarking tool is een uitbreiding van YCSB[8] met (1) ondersteuning voor het uitvoeren van instructies op gegeven tijdstippen en (2) een implementatie om leesbewerkingen uit te voeren tijdens en onmiddellijk na het schrijven van data. De eerste uitbreiding is gebruikt voor de beschikbaarheidstesten, de tweede voor de consistentietesten. Deze benchmarking tool biedt basistesten aan die conclusies mogelijk maken. Zoals vermeldt in het verder werk kan deze testmethode verder uitgebreid worden.

Analyseren van verschillende DBMS's In hoofdstuk 4 zijn de observaties van drie verschillende DBMS's aan bod gekomen. Alle systemen zijn getest met behulp van de beschikbaarheidstesten, HBase en MongoDB zijn ook getest met de consistentietesten. In hoofdstuk 5 zijn de resultaten bestudeerd en de verschillen tussen de systemen besproken.

Uit de beschikbaarheidstesten blijkt dat de gecentraliseerde aanpak van Pgpool-II keer op keer dezelfde reactie geeft. De reacties van HBase en MongoDB zijn verschillend tussen verschillende uitvoeringen van dezelfde test. Beide systemen ondersteunen, in tegenstelling tot Pgpool-II, automatisch herstel van een node in het

6. CONCLUSIE

gedistribueerde opslag systeem.

Bij de consistentietesten is er een verschil hoe gelijktijdige bewerkingen worden uitgevoerd in MongoDB en HBase. Bij HBase wordt de schrijflock pas vrijgegeven na het volledig voltooien van de schrijfbewerking, hierna kan de leesbewerking uitgevoerd worden. In MongoDB wordt de schrijflock al vrijgegeven voor het volledig voltooien van de bewerking, met als gevolg dat leesacties de nieuwe data al lezen vooraleer een schrijfactie volledig voltooid is.

Bepaalde reacties van de systemen kunnen nog niet verklaard worden, maar met meer onderzoek is het mogelijk dat de reden achterhaald zou kunnen worden. Maar voor toekomstige gebruikers van deze DBMS's is er nu een overzicht wat de eigenschappen en het gedrag is bij het uitvallen van een service, node in het gedistribueerde opslag systeem of netwerk verbinding en hoe gelijktijdige lees- en schrijfbewerkingen afgehandeld worden.

Eenvoudig herhalen en uitbreiden van de testen Om de testen eenvoudig te kunnen herhalen, is de volledige testinfrastructuur op te stellen met behulp van IMP. Er is geen kennis nodig van de database systemen of de testsoftware, enkel de installatie van IMP met de nodige modules is nodig, gevolgd door het opstellen van de gewenste staat in een configuratie bestand en uiteindelijk de uitrol d.m.v. IMP. De testen kunnen uitgebreid worden naar andere DBMS's, dit gebeurt in eerste instantie door de ondersteuning in YCSB te controleren of te implementeren, er is al standaard ondersteuning voor vele DBMS's. Daarna dienen de testen geconfigureerd worden specifiek voor deze database met behulp van de kalibratie. Daarna kunnen de beschikbaarheids- en consistentietesten voor dit nieuw systeem uitgevoerd worden.

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 dat geïnspireerd is door 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 laatst geschreven waarde in geval van een conflict zal opgeslagen worden.

A. BESPREKING VAN VERSCHILLENDEN DBMS's

De database kan gedistribueerd uitgerold worden. 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 mogelijkheden.

A.1.2 HBase

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

HBase is een database systeem dat gebaseerd is op Google's BigTable en gebruik maakt van Zookeeper en 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 HRegions. Een HRegionServer is een verantwoordelijk voor de data van bepaalde HRegions. 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 CouchDB

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

Apache CouchDB is een document database systeem waar alles wordt voorgesteld de interactie verloopt met behulp 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. Het is 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 verder te kunnen werken.

In een gedistribueerde omgeving ziet CouchDB conflicten niet als een uitzondering maar als een normale omstandigheid. Updates zullen atomisch op rijbasis 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.

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 die door de servers zelf wordt gekozen en opgevolgd. 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 kiezen.

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 voor het 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 uit te voeren. Redis biedt ook ondersteuning voor transacties en heeft 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 schrijfoperatie weggeschreven en kan de database opnieuw opgebouwd worden met behulp van deze lijst.

Tenslotte heeft Redis momenteel een 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. Er is ook de mogelijkheid tot master-slave opstelling met automatische failure detection. Dit laatste is nog wel in beta, al is het mogelijk om deze functie te gebruiken. Tenslotte is er in de toekomst meer ondersteuning op komst met behulp van Redis Cluster waar data automatisch verdeeld 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 objecten. Daarnaast ondersteunt de database de standaard operaties zoals het invoegen, lezen, aanpassen en verwijderen met enkele uitbreidingen hierop. Zo is het mogelijk om secundaire indexen te definiëren op de elementen, MapReduce toe te passen en een full-text search uit te voeren.

Riak is gebouwd om in gedistribueerde omgeving te werken waar instanties gelijkaardige functies hebben. 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 waardoor 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 modi van distributie. De opbouw bestaat uit verschillende lagen, elk met hun eigen gedefinieerde functie. Met behulp van deze lagen kan de ontwikkelaar de functionaliteit aanpassen door het toevoegen van een nieuwe laag. Data wordt verdeeld met behulp van consistente hashing over de verschillende servers. Hierbij wordt data verschillende keren opgeslagen om ervoor te zorgen dat de data nog beschikbaar is in het geval van falen van een enkele instantie.

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 door het gebruik van SQL.

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 slaves, zal de database verder werken. Bij het falen van de master zal een nieuwe master handmatig aangeduid moeten worden. Ook de recovery moet handmatig opgestart worden, waarna de originele master opnieuw als master kan gezet worden indien dit gewenst is(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 met behulp van SQL.

Er is een groter verschil als de database ook gedistribueerd moet uitgerold worden. De standaard ondersteuning van PostgreSQL in een gedistribueerde omgeving is net zoals bij MySQL een Master-Slave configuratie. Maar deze ondersteuning kan uitgebreid worden door het gebruik maken van externe modules. Er bestaan verschillende modules, 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 modi, zoals parallel mode waar de data verdeeld wordt over verschillende instanties. De replicatie mode zorgt ervoor dat data op meerdere instanties wordt opgeslagen zodat de data nog beschikbaar is bij het falen van een enkele instantie.

Bijlage B

Overzicht van gedetailleerde implementatie keuzes

In dit hoofdstuk bevinden zich 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

Naam	eenheid	Omschrijving
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.

Naam	eenheid	Omschrijving
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

Stoppen	
Wat	Commando
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

Heropstarten	
Wat	Commando
Zachte start	service {{service-name}} restart
Harde start	service {{service-name}} restart
Netwerk herstellen	iptables -D OUTPUT 1

Speciale commando's	
Wat	Commando
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

Extern beschikbare code en resultaten

Dit hoofdstuk bevat een overzicht van de locaties van de externe code en resultaten.

- **IMP HBase** (<https://github.com/thuys/HBase>): De installatie en configuratie van HBase met behulp van IMP.
- **IMP MongoDB** (<https://github.com/thuys/mongodb>): De installatie en configuratie van MongoDB met behulp van IMP.
- **IMP Pgpool-II** (<https://github.com/thuys/postgresql>): De installatie en configuratie van Pgpool-II (PostgreSQL) met behulp van IMP.
- **IMP YCSB** (<https://github.com/thuys/ycsb>): De installatie en configuratie van YCSB met behulp van IMP.
- **YCSB code** (<https://github.com/thuys/YCSB-Implementation>): De aangepaste code van YCSB.
- **YCSB Analyse code** (<https://github.com/thuys/YCSB-R-Scripts>): De R code om de YCSB analyse op uit te voeren.
- **Ruwe testdata** (<https://github.com/thuys/YCSB-Testdata>): De ruwe testdata met grafieken gebruikt voor de thesis.

Bijlage D

Paper: "CAP in practice: HBase and MongoDB"

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

CAP in practice: HBase and MongoDB

Thomas Uyttendaele (thomas.uyttendaele@student.kuleuven.be)
Departement Computerwetenschappen, KULeuven, België

Abstract—This paper introduces a new, quantified method to test distributed database systems 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, some interesting outcomes are described in this paper.

I. INTRODUCTION

New online services and more online users means more data, data and load a single server can't handle. Database systems are transformed towards a more distributed approach, for higher availability in case of an unexpected crash but also for horizontal scalability: the data of a single database is spread over different systems. Several new software have other requirements on the database than a few years ago; e.g. a change in a record doesn't need to be read correctly by other users immediately, this can happen in time. This behaviour is called eventual consistency.

Over the past years, many new systems have been built on this wave of changes, categorized as NoSQL. Some applications contain a high volume of data, have the need for consistent data and need 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 these systems will be described. In chapter II a brief overview of the CAP Theorem is given, chapter III discusses HBase and MongoDB based on the literature. 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 formulated in chapter VIII.

II. THE CAP THEOREM[1][2]

The CAP Theorem was introduced by E. Brewer [1] in 2000 and discusses three properties of which each network shared-data system can guarantee at most two: (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 systems and sometimes it

was misused. As E. Brewer explains 12 years 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 there are 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 or 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 built based on the two phase commit and have cache invalidation protocols. Examples of this type 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 some nodes during the partition 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 built 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 system with partition tolerance. It is 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. 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. 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 is guaranteed due to the fact that there is a single system responsible for the storage of a record. 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 reply 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 that are 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 and dividing the data between them.

A ReplicaSet exists out of different MongoDB servers whom work as master and slaves. 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 liveness 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. HBase has 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-writer support 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 in YCSB 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. These tests contains 3 kinds of tests, of which an individual test takes 900 second. The first action takes place at 300 seconds, the second at 600 seconds.

- (1) Graceful shut down of a data service and (2) restart of the service
- (1) Hard kill of the data service and (2) a restart of the service
- (1) Blocking of all network traffic from and to a server and (2) 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 or a server crashes.

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 a user-defined amount of readers. Each writer will insert or update a value in the database at a user-configurable 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.

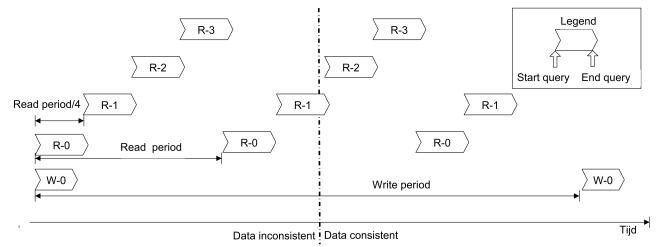


Fig. 1. Example of consistency support for one write period with 1 writer and 4 readers

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

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 in MongoDB. In 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 order of the reading and updating of the records is defined by a Zipfian¹ distribution. 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

¹Some record are popular, others rare to be used.

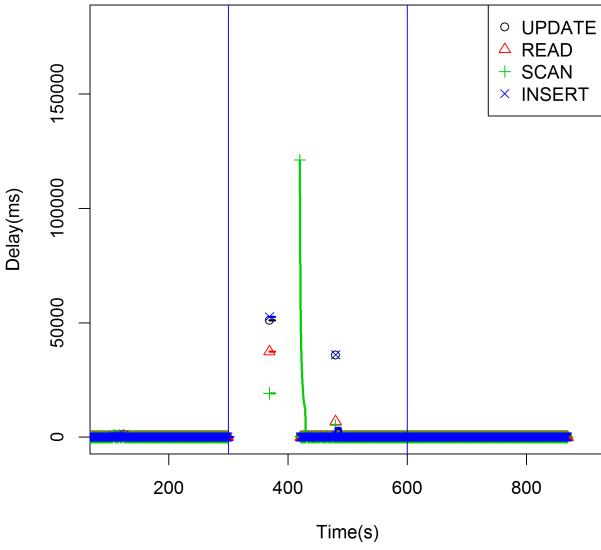


Fig. 2. Example of network partition. The requests block for 110 seconds

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, no queries can be executed on the database connection till the server is brought back online. In this scenario it was needed to manually disconnect and reconnect to the database. A reason why this happens is still unknown.

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 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. It goes from no influence, a temporary unavailability for a few seconds, or no completed queries the whole time the server is down. If all connections are reinitialised for the last case, new queries can be executed. If 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

Based on 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 inserting or updating the value, the read query will wait till the completion of the write query and return immediately the correct value. If a read query is sent too soon, it will return the old data. Each query that is finished 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.

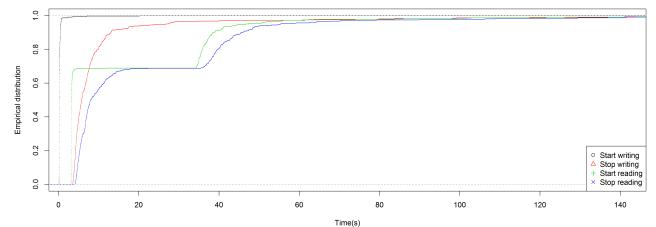


Fig. 3. Consistency: cumulative distribution of queries that read the correct value if reading is started 3ms, 13ms, 23ms. Measurement of over 400 values.

MongoDB: A read query can return the old or new data before the write query has been completed. Despite of the use of a readers/writer lock mechanism, 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 consistency window for the different readers if the start moments of the writers and the successful read queries are compared. The difference is the guarantee a user has when a write query has finished.

An overview of the consistency windows is shown in figure 4 and 5 for respectively HBase and MongoDB.

VI. FUTURE WORK

The test executed on the different systems are a start to have a quantified approach on the consistency and availability of the different systems. Besides testing more systems, deeper testing can be done and extensions can be added to the benchmarking tool.

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

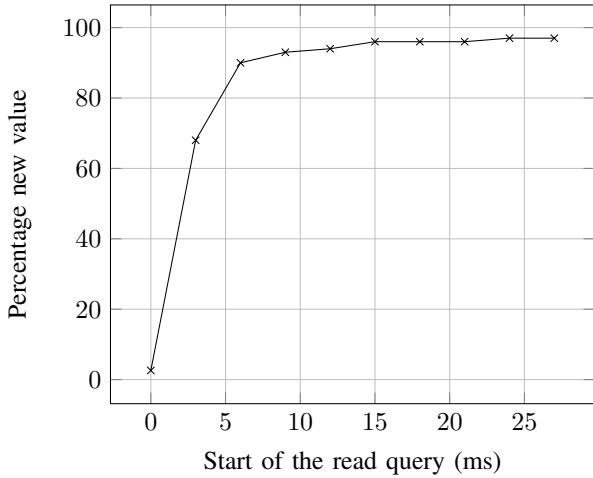


Fig. 4. Consistency: Percentage of the queries started at the given time moment that will read the newest inserted data. The average latency in a random read transaction is around 6ms.

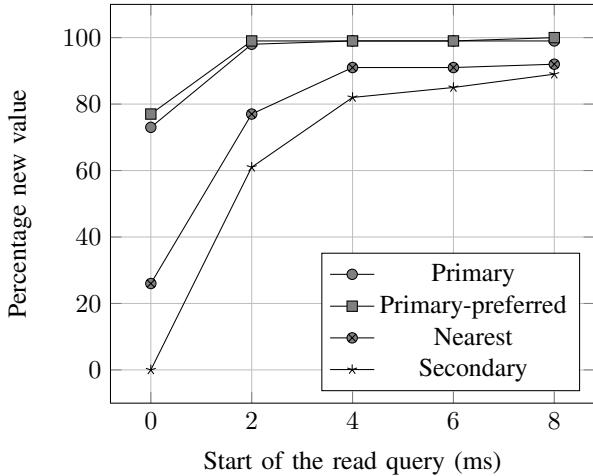


Fig. 5. Consistency: Percentage of the queries started at 0, 2, 4 and 8ms that will read the newest inserted data with each row for MongoDB. The average latency on a read transaction is for all read queries around 1ms.

tolerance. How is the availability of each node? How long is a primary in MongoDB still accepting queries after it has been cut off from the other instances?

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 to come up with a mathematical formula which could predict the average consistency window for example.

VII. RELATED WORK

The is only a limited amount of research executed towards the guarantees a database system provides towards consistency and availability. 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 the current research and describe a missing category of testing methods towards consistency. This category is called the passive analysis, this analysis focusses on how a database user observes the consistency guarantees. This is different from the active analysis where they mostly focus on the time it takes to replicated to all nodes. With the testing method described in this paper, both are covered: 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. These caches are by default enabled on the client side, but the submission of the client cache towards the system does not only depend on the time. Also the amount of traffic generated at the client influences this speed. In the study they compare different cache sizes and this shows already a difference in time. In case data needs to be strict consistent, it is better to disable caching at this moment, this way the behaviour is more predictable and in general also faster available for other clients.

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. In some occasions both systems 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 manual.
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.
- [8] W. Golab, M. R. Rahman, A. AuYoung, K. Keeton, X. S. Li, Eventually consistent: not what you were expecting?, Communications of the ACM 57 (3) (2014) 38–44.
- [9] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, B. Rinaldi, 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.
- [10] J. Mauro, J. Zhu, I. Pramanick, The system recovery benchmark, in: Dependable Computing, 2004. Proceedings. 10th IEEE Pacific Rim International Symposium on, IEEE, 2004, pp. 271–280.

Bijlage E

Poster: ”CAP in de praktijk: MongoDB”

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



KATHOLIEKE UNIVERSITEIT
LEUVEN

FACULTEIT
INGENIEURSWETENSCHAPPEN

Master
Computer-
wetenschappen

Masterproef
*Thomas
Uyttendaele*

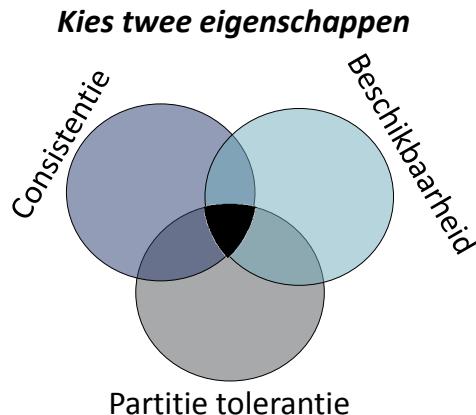
Promotor
*Professor Wouter
Jootsen*

Academiejaar
2013-2014



CAP in praktijk: MongoDB

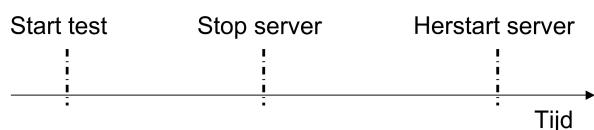
Het CAP Theorema



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

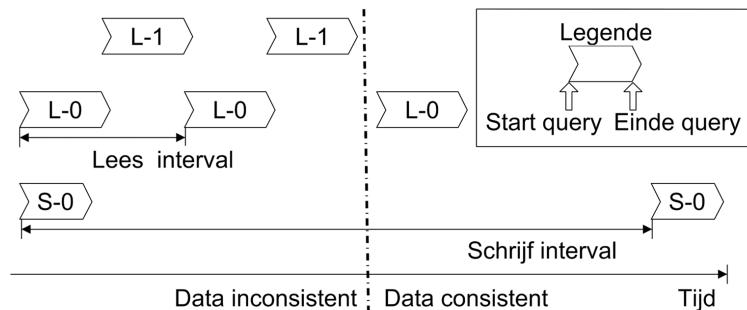


Beschikbaarheid

- Op 300s: Stop server
- Op 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

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%

Partitie tolerantie

- Partitie <50% van servers**
 - Onbeschikbaar voor schrijven en lezen

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. 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. 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 (bezocht op 03-02-2014).
- [34] Tilman 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/bescofield/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: Evaluatie van consistentie en beschikbaarheid in gedistribueerde database systemen

Engelse titel: Evaluation of consistency and availability for distributed database systems

UDC: 681.3

Korte inhoud:

In de database wereld zijn er momenteel twee grote categorieën van database systemen actief: de relationele en NoSQL systemen. De laatste categorie maakt verschillende keuzes in het CAP theorema en heeft uit een grote diversiteit aan datamodellen en performantie-eigenschappen.

Deze thesis focust op de uitwerking van een benchmarking tool om consistentie en beschikbaarheid in een gedistribueerde omgeving te testen.

De beschikbaarheidstesten onderzoeken het gedrag van het gehele database systeem bij het in- en uitschakelen van een service, een node in het gedistribueerde opslag systeem of netwerk verbinding. De consistentietest bestudeert hoe het systeem omgaat met gelijktijdige schrijf- en leesbewerkingen op hetzelfde datarecord.

Deze benchmark onderzoekt het gedrag van een database systeem in de praktijk en vergelijkt dit met de documentatie. Daarnaast worden verschillende database systemen met elkaar vergeleken.

Drie verschillende database systemen worden getest met behulp van de benchmarking tool, waarna de resultaten geanalyseerd worden. Op basis van deze resultaten komt het verschillend gedrag tussen de systemen tot uiting zoals op een verschillende wijze omgaan met gelijktijdig lezen en schrijven van een data-element.

De installatie en configuratie van de benchmarking tool en de reeds geteste database systemen is geautomatiseerd om te kunnen opstellen en uitvoeren zonder voorkennis van de specifieke systemen. Met een beperkte kennis en kost kan de tool uitgevoerd worden op andere database systemen.

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

Assessoren: Dr. Tias Guns

Dr. ir. Christophe Huygens

Begeleiders: Dr. ir. Bart Vanbrabant

Dr. Bert Lagaisse