KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Automatisch uitrol van database systemen en vergelijking van beschikbaarheid

Thomas Uyttendaele

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

Promotor:
Prof. dr. ir. Wouter Joosen

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

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

Academiejaar 2013 – 2014

Voorwoord

Sed feugiat. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Ut pellentesque augue sed urna. Vestibulum diam eros, fringilla et, consectetuer eu, nonummy id, sapien. Nullam at lectus. In sagittis ultrices mauris. Curabitur malesuada erat sit amet massa. Fusce blandit. Aliquam erat volutpat. Aliquam euismod. Aenean vel lectus. Nunc imperdiet justo nec dolor.

Voorwoord schrijven

Etiam euismod. Fusce facilisis lacinia dui. Suspendisse potenti. In mi erat, cursus id, nonummy sed, ullamcorper eget, sapien. Praesent pretium, magna in eleifend egestas, pede pede pretium lorem, quis consectetuer tortor sapien facilisis magna. Mauris quis magna varius nulla scelerisque imperdiet. Aliquam non quam. Aliquam porttitor quam a lacus. Praesent vel arcu ut tortor cursus volutpat. In vitae pede quis diam bibendum placerat. Fusce elementum convallis neque. Sed dolor orci, scelerisque ac, dapibus nec, ultricies ut, mi. Duis nec dui quis leo sagittis commodo.

Aliquam lectus. Vivamus leo. Quisque ornare tellus ullamcorper nulla. Mauris porttitor pharetra tortor. Sed fringilla justo sed mauris. Mauris tellus. Sed non leo. Nullam elementum, magna in cursus sodales, augue est scelerisque sapien, venenatis congue nulla arcu et pede. Ut suscipit enim vel sapien. Donec congue. Maecenas urna mi, suscipit in, placerat ut, vestibulum ut, massa. Fusce ultrices nulla et nisl.

Thomas Uyttendaele

Inhoudsopgave

Todo list

Samenvatting

Abstract schrijven

In dit `abstract` environment wordt een al dan niet uitgebreide samenvatting van het werk gegeven. De bedoeling is wel dat dit tot 1 bladzijde beperkt blijft. Sed feugiat. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Ut pellentesque augue sed urna. Vestibulum diam eros, fringilla et, consectetuer eu, nonummy id, sapien. Nullam at lectus. In sagittis ultrices mauris. Curabitur malesuada erat sit amet massa. Fusce blandit. Aliquam erat volutpat. Aliquam euismod. Aenean vel lectus. Nunc imperdiet justo nec dolor.

Etiam euismod. Fusce facilisis lacinia dui. Suspendisse potenti. In mi erat, cursus id, nonummy sed, ullamcorper eget, sapien. Praesent pretium, magna in eleifend egestas, pede pede pretium lorem, quis consectetuer tortor sapien facilisis magna. Mauris quis magna varius nulla scelerisque imperdiet. Aliquam non quam. Aliquam porttitor quam a lacus. Praesent vel arcu ut tortor cursus volutpat. In vitae pede quis diam bibendum placerat. Fusce elementum convallis neque. Sed dolor orci, scelerisque ac, dapibus nec, ultricies ut, mi. Duis nec dui quis leo sagittis commodo.

Aliquam lectus. Vivamus leo. Quisque ornare tellus ullamcorper nulla. Mauris porttitor pharetra tortor. Sed fringilla justo sed mauris. Mauris tellus. Sed non leo. Nullam elementum, magna in cursus sodales, augue est scelerisque sapien, venenatis congue nulla arcu et pede. Ut suscipit enim vel sapien. Donec congue. Maecenas urna mi, suscipit in, placerat ut, vestibulum ut, massa. Fusce ultrices nulla et nisl.

Lijst van figuren en tabellen

Lijst van figuren

Lijst van tabellen

Lijst van afkortingen en symbolen

Afkortingen

IMP	Infrastructure Management Platform
DBMS	Databasemanagementsysteem
RDBMS	Relationeel Databasemanagementsysteem
Range Query	Het opvragen van een set van records met behulp van een enkele query
BASE	
ACID	zer

Symbolen

42	aaa

Hoofdstuk 1

Inleiding

Hoofdstuk 2

Overzicht van de technologie

De huidige state of the art database systemen zijn er gekomen na een evolutie doorheen de tijd. Dit hoofdstuk bespreekt eerst de geschiedenis van de database waarna er in meer detail de 2 belangrijkste categorieën aanbod komen: de relationele en NoSQL databases. Daarna zullen van beide categorieën enkele systemen in meer detail besproken worden.

Tenslotte wordt dit hoofdstuk afgerond met een vergelijking van de huidige methodes om de databases op een objectieve manier te vergelijken.

2.1 Geschiedenis van de databasemanagementsystemen

Doorheen de geschiedenis heeft de mens verschillende manieren gebruikt om data op te slaan, te verwerken en terug te vinden. Hiervoor zijn er verschillende stappen van data management geweest, tot voor het ontstaan van de computer ging dit met pen en papier of met ponskaarten[15]. Met de opkomst van de computer, werden nieuwe methodes gebruikt die zijn mee geëvolueerd met de vooruitgang in de technologie en de veranderingen in het gebruik van de data. De hiervoor ontwikkelde software wordt gecategoriseerd onder het **databasemanagementsysteem**(DBMS's). De ontwikkeling en opkomst van de DBMS's kan in verschillende fasen opgedeeld worden.

De eerste DBMS's zijn er gekomen met de introductie van de mainframes zoals UNIVAC1 en de ontwikkeling van specifieke programmeertalen voor het werken met deze data, onder andere conferenties zoals CODASYL hebben de ontwikkeling van COBOL en andere standaarden mee ontwikkeld[15].

De grote verandering in DBMS's is er gekomen door het artikel van E. Codd over het relationele model in 1969 [8]. Het sleutel concept van het relationele model is dat de data georganiseerd is in relaties (tabellen) die gekoppeld zijn door middel van keys (constraints), hierbij wordt zoveel mogelijk redundante data vermeden. Voorbeelden

van populaire RDBMS's (relationele DBMS's) zijn Oracle, MySQL en PostgreSQL. Meer informatie komt aanbod in de volgende sectie.

De laatste nieuwe generatie zijn de NoSQL databases die sinds 2000 zijn begonnen, NoSQL staat voor '*Not only SQL*'. Deze systemen zijn er gekomen als reactie op het relationele model en willen meer flexibele database, lagere complexiteit, hogere doorvoer van data, horizontale schaalbaarheid en het draaien op commodity hardware. Verschillende voorbeelden van NoSQL systemen zijn Google BigTable, Amazon Dynamo, HBase, MongoDB, ... [37] Meer informatie en een vergelijking met relationele databases komt aanbod in de volgende sectie.

2.2 Relationele en NoSQL DBMS's

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

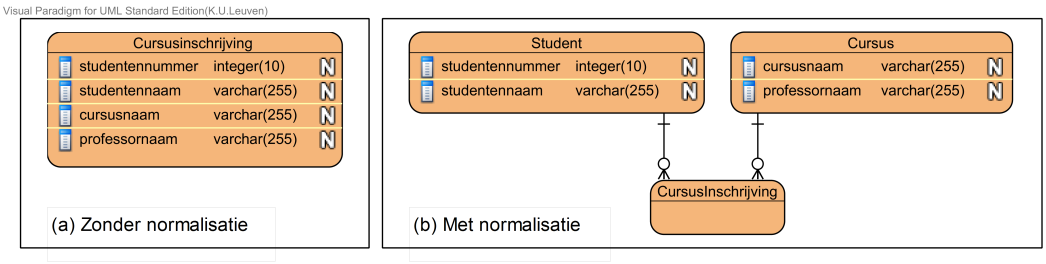
2.2.1 Relationele database

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

Het relationele model is gebaseerd op de theoretische wiskundige principes van set-theorie en eerste-orde predicaten logica. Het model organiseert de data in tabellen en relaties tussen de tabellen. De tabel heeft kolommen die verschillende velden voorstellen en elke rij een collectie van gerelateerde datawaardes is. De relaties tussen de verschillende tabellen toont mogelijke connecties. 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.[11]
De normalisatie kan geïllustreerd worden met het korte voorbeeld in figuur 2.1: de professor voor een vak zal bij elke student hetzelfde zijn, het veranderen van een professor voor een vak zou in het eerste geval een update van alle ingeschreven studenten inhouden, in het tweede geval is dit maar de aanpassing van een enkel record, hetzelfde geldt voor de student.
Interactie met de RDBMS gebeurt op basis van SQL (Structured Query Language), een taal gebaseerd op de relationele logica geeft uitgebreide query mogelijkheden aan de gebruiker van de software.

Een belangrijk concept in een relationele database is ACID, welk tussen verschillende transacties wordt gegarandeerd:

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



Figuur 2.1: Relationeel datamodel (a) zonder en (b) met normalisatie

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

Geïsoleerd (Isolation) Een transactie moet uitgevoerd worden alsof ze geïsoleerd is van andere transacties, die eventueel gelijktijdig uitgevoerd worden.

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

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

2.2.2 NoSQL database

NoSQL DBMS zijn ontstaan als reactie op de 'one size fits all'-gedachte die RDBMS's volgen. Dit is ook zichtbaar in de NoSQL systemen, ze bestaan in verschillende variëteiten, elk met hun eigen eigenschappen en toepassingsgebied. Toch is er een rode draad te vinden tussen de verschillende systemen vergeleken met RDBMS:

- **Lagere complexiteit**: NoSQL systemen bieden minder opties en features aan dan de RDBMS omdat bepaalde applicaties enkelen nu eenmaal niet nodig hebben. Bijvoorbeeld in een sociale netwerk moet een post niet onmiddellijk beschikbaar zijn voor al de vrienden van een persoon, maar dit mag even duren.
- **Hogere doorvoer**: Talrijke NoSQL systemen bieden een hogere doorvoer van data aan, meestal gecombineerd met een lagere complexiteit.

- **Horizontale schaalbaarheid en werkend op commodity hardware**: Waar grote RDBMS's draaien op dure high-end systemen, was het bedoeling van NoSQL databases om te werken met eenvoudige machines (commodity hardware).
  Horizontale schaalbaarheid staat voor het toevoegen extra machines aan een systeem voor extra resources, in tegenstelling tot verticale schaalbaarheid waar een krachtiger machine wordt gebruikt voor de opschaling. In horizontale opschaling wordt de opschaling gedaan 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 of meer gelijkend maken dan RDBMS's. Waar dat RDBMS nog zijn ontworpen voor de object georiënteerde programmeertalen, werd er bij de ontwikkeling van NoSQL onmiddellijk hiermee rekening gehouden.

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

- Basis beschikbaarheid (**B**asically **Availability**): het DBMS probeert de data nog steeds beschikbaar te houden, zelf in het geval van één of meerdere falende instanties.

- **S**oft State: De data moet op een bepaald moment niet volledig consistent zijn.

- Eventuele consistentie (**E**ventual 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 update onmiddellijk correct lezen.
  - *Session* consistentie: De gebruiker zal zijn updates onmiddellijk kunnen lezen binnen dezelfde sessie, een sessie is hierdoor meestal gelimiteerd tot een enkele database server.
  - *Casual* consistentie: Als een gebruiker versie X leest en vervolgens versie Y schrijft, zal elke gebruiker die versie Y leest ook versie X lezen.
  - *Monotonic Read* consistentie: Dit levert monotone tijdsgaranties dat een gebruiker enkel recentere data versies in de toekomst zal lezen.

Deze vereisten kunnen gekoppeld worden aan de CAP theorie van Erik Brewer[4]. CAP zegt dat een gedistribueerd systeem maar twee van de 3 CAP elementen kan ondersteunen: (strikte) consistentie, beschikbaarheid en partitie tolerantie. Met de laatste wordt bedoelt dat enkele instanties van het systeem falen maar er nog steeds een werkend systeem 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 2.1.

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

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

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

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.

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]

2.3 Bespreking van verschillende DBMS's

De verschillende categorieën databases zijn besproken, voor deze thesis zullen enkele relationele en NoSQL DBMS's in meer detail worden besproken. Databases uit 4 categorieën komen verder aanbod, er is gekozen om de Graph NoSQL DBMS's niet te bespreken omdat deze gericht zijn op andere dataset dan de overige categorieën.

- 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
- Relationele DBMS's: MySQL, Pgpool-II (PostgreSQL)

2.3.1 Column database

Cassandra

Website: *http://cassandra.apache.org/*
Cassandra is een database systeem die gebaseerd is op 2 verschillende systemen, Amazon's Dynamo en Google's Bigtable, wat voor een combinatie van een column- en key-value-based database zorgt.

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

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

HBase

Website: *http://hbase.apache.org/*
HBase is een database systeem die gebaseerd is op Google's BigTable en draait boven op HDFS, Hadoop Distibuted File System.

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

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

2.3.2 Document database

Apache CoucheDB

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

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

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

MongoDB

Website: http://www.mongodb.org/
MongoDB is een document database systeem waar de data wordt voorgesteld aan de hand van BSON, een binaire vorm vergelijkbaar met JSON. Data kan ingegegeven worden via JSON aangezien er een eenvoudige map mogelijk is.

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

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

2.3.3 Key-Value database

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.

MemCacheDB

Updaten

Website: http://memcachedb.org/
MemCacheDB is een veel gebruikt systeem waarin al de data in RAM geheugen wordt gehouden en alhoewel er ondersteuning is met behulp van MemCacheDB voor persistentie is deze database niet bedoeld voor persistente opslag.

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.

Redis

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

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

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

in de toekomst meer ondersteuning op komst met behulp van Redis Cluster waar data automatisch verspreid wordt over verschillende instanties.

Riak

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

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

Project Voldemort

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

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

2.3.4 Relationele database

MySQL

Website: *http://www.mysql.com/*
MySQL is een relationele database waarin data kan voorgesteld worden in verschillende vormen, beginnend met een bool tot een blok tekst. Daarnaast zijn de query mogelijkheden uitgebreid.

De uitbreiding van een gedistribueerd systeem is bij MySQL ingebouwd door middel van een Master-Slave configuratie. Als mysqlfailover een faal detecteert in één van de slaven, zal de database verder werken, bij het falen van de master zal een nieuwe master handmatig aangeduid moeten worden. Ook de recovery moet handmatig

gestart worden, waarna indien gewenst de originele master opnieuw als master kan gezet worden (bv. omdat deze de krachtigste computer is).

Pgpool-II (PostgreSQL)

Website: http://www.pgpool.net/
PostgreSQL is een relationele database en heeft soortgelijke specificaties als MySQL op een enkele computer, verschillende soorten data kunnen voorgesteld worden met uitgebreide query mogelijkheden.

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

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

2.3.5 Conclusie

Deze 10 systemen laten een variatie van verschillende opties zien, zowel in query mogelijkheden als naar een gedistribueerde configuratie waar er keuzes volgens het CAP theorema gemaakt moeten worden.

Bij het kiezen van een systeem is het moeilijk om een keuze te maken af en toe, in het volgend gedeelte zal er besproken worden wat er momenteel beschikbaar is als resultaten of te gebruiken tool.

2.4 Objectieve vergelijking van de verschillende systemen volgens CAP

De databasemanagementsystemen zijn voor meer dan 40 jaar actief. Met de opkomst van het relationele model in 1969 door E. Codd [7], zijn gedurende vele jaren de relationele DMBS de leidende technologie geweest. Maar de afgelopen 10 jaar is er in het landschap veel veranderd met de opkomst van de NoSQL DBMS's die afstappen van het ACID naar BASE, meer gefocust op het werken op grote data in een gedistribueerde omgeving.

Nu zijn er vele verschilpunten tussen verschillende systemen, dit gaat van het datamodel tot de verschillende keuzes in het CAP theorema. In dit gedeelte zal er gekeken worden welke methodes er al beschikbaar zijn voor het meten van de performantie, consistentie en beschikbaarheid en mogelijke resultaten.

2.4.1 Performantie benchmarking

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

Als eerste hebben vele DBMS's interne benchmarking tools, waarmee de database op verschillende configuraties kan getest worden en vergeleken worden. Deze resultaten zijn nuttig indien het DBMS al gekozen is en men bezig is met het aanpassen van de parameters of om te onderzoeken wat net de bottleneck is in een bepaald systeem. Een voorbeeld hiervan is mongoperf[1] voor MongoDB.

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

De eerste categorie, het **ontwikkelen van een tool**, heeft als grote voordeel dat andere gebruikers nadien de testen opnieuw kunnen uitvoeren met de huidige systemen. Het is namelijk niet gegarandeerd dat het resultaat van een jaar geleden met de nieuwste versie nog te vergelijken is. Het grootste nadeel is het type testen dat kan 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 de opensource tool YCSB[9]. In deze tool kan elk DBMS's getest worden zolang een basisset van 5 queries kan implementeren: het invoegen, updaten, verwijderen en opvragen van een enkel record met daarnaast ook de mogelijkheid tot scan queries, met behulp van 1 query een verzameling van records tegelijk op te vragen.
Sommige systemen ondersteunen bepaalde queries niet onmiddellijk maar bevatten wel de functionaliteit om deze via een omweg te implementeren, bijvoorbeeld een update kan geïmplementeerd worden door het opvragen, verwijderen en vervolgen invoegen van de geüpdatete record.

Een volgende categorie zijn de **resultaten van gerelateerde DBMS's**, in deze categorie zijn er voornamelijk resultaten te vinden van 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. Hoewel in de categorisatie van deze thesis de eerste twee onder column NoSQL vallen, zijn

[1] http://docs.mongodb.org/manual/reference/program/mongoperf/

deze nog vrij gelijklopend.
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 zoals appels en appelsienen.

Als laatste categorie, zijn er de **resultaten van verschillende DBMS's** waar zeer verscheidene 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 ook hier dat er een gemeenschappelijke subset gevonden moet worden, hierdoor kunnen bepaalde databases hun kracht net niet laten zien. Verschillende van deze onderzoeken zijn [38] en [34]. Deze laatste maakt gebruik van de YCSB tool die hierboven besproken was.

2.4.2 Consistentie testen

Zoals besproken in het vorige gedeelte, is er al relatief veel onderzoek gebeurd naar de performantie van de verschillende systemen. Maar daarnaast is er ook de consistentie eigenschappen die verschillend kunnen zijn.

Een recent artikel [14] (maart 2014), begint met het stellen dat er momenteel nauwelijks gekwantificeerde methodes bestaan om de eventuele consistente te meten. In hun artikel stellen zijn mogelijke methoden voor: de actieve of passieve analyse.
De **actieve** analyse bestaat het wegschrijven van data in een database waarna 1 of meerdere andere gebruikers meten hoe lang het duurt vooraleer zij de nieuwe waarde lezen. De **passieve** analyse wordt er gekeken hoe de gebruiker interageert met het systeem en hoe de data updates zich gedragen. Leest deze altijd de laatste waarde (=strikte consistentie)? Is het mogelijk dat een nieuwe waarde al wordt gelezen voor de schrijfactie voltooid is?
Beide systemen hebben hun eigenschappen, de actieve analyse kan gezien worden als een systeem georiënteerde analyse en test hoe lang het duurt voor de data beschikbaar is over de verschillende systemen en heeft als uitkomst hoe DBMS's zich verschillend gedragen ten opzichte van elkaar of in verschillende netwerk- en hardwareomgevingen. Bij de passieve analyse is georiënteerd naar de gebruiker toe, hoe moet deze zijn toepassingen aanpassen, wat zijn specifieke eigenschappen van het systeem?

Voor beide analyse methodes is er al onderzoek gebeurd, maar het meeste is gebeurd naar de actieve analyse. Onder andere Duitse onderzoekers hebben op het Amazon S3 platform getest hoe lang het duurt vooraleer data geschreven in MiniStorage, een database systeem, beschikbaar is voor alle gebruikers. [1].
Daarnaast zijn er ook 2 interessante resultaten gevonden: allereerst heeft het Amazon

S3 systeem geen monotone lees consistentie, daarnaast bleek het inconsistentie interval voor een bepaald record periodiek verloop te hebben dat niet door de onderzoekers verklaard konden worden.

De YCSB software van hierboven is door onderzoekers in de VS uitgebreid naar YCSB++[28] waardoor deze meer ondersteuning heeft om het meten van systeembelasting maar ook voor de consistentie eigenschappen. Hoewel enkele van de systemen die zij testen in principe strikt consistent zijn zoals HBase, worden deze eventueel consistent door het gebruiken van buffers bij de gebruiker. Vervolgens testen zij hoe lang het duurt voor de data ook gelezen kan worden. Een probleem met deze methode is dat deze vertraging sterk afhankelijk is van het aantal acties van de schrijvende gebruiker: indien er meer geschreven wordt, zal de buffer sneller verzonden worden naar de server en dus sneller beschikbaar zijn voor andere gebruikers.
Hoewel zij stellen dat er ook testen zijn gedaan naar eventuele consistentie voor Cassandra en MongoDB, zijn de resultaten niet beschikbaar in het artikel of op de website.

Andere onderzoeker[40] hebben passieve analyse uitgevoerd en gekeken hoe lang het duurt vooraleer databases in de cloud infrastructuur consistente data hebben, met een focus op Amazon SimpleDB. Er zijn testen uitgevoerd hoe lang het duurt vooraleer de meest recente data gelezen zal worden door een gebruiker.

Bij Netflix heeft men aan passieve analyse gedaan op hun Cassandra systeem [20] waar zij in hun testen geen consistentie problemen 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.

2.4.3 Beschikbaarheidstesten

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

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

Voor informatie over hoe de verschillende systemen reageren, is het momenteel

uitzoeken op de website van de software verdeler en zelf uittesten van het gedrag. Naar mijn onderzoek, bestaat er nog geen vergelijkende studie tussen de verschillende systemen.

2.5 Conclusie

Na het artikel van E. Codd [7] in 1969, heeft het relationele model een dominante rol gespeeld in het database model. Met de opkomst van het internet, grotere hoeveelheden data en steeds complexere RDBMS's, is er een nieuwe stroming gekomen in de database wereld, de NoSQL DBMS's. Deze beloven betere schaalbaarheid, hogere performantie en dit op commodity hardware ten aanzien van de *ACID* eigenschappen naar *BASE*: een hogere beschikbaarheid en eventuele consistentie.

In deze thesis zal er eerst een algemene methode voorgesteld worden om verschillende systemen te testen naar de eventuele consistentie, behandelen van storingen (failure handling) en (automatisch) herstel (automatic recovery).
Daarna zal deze methode uitgevoerd worden op verschillende databases waar verschillende resultaten en opvattingen gezien kunnen worden. Deze leiden tot enkele initiële conclusies die in de toekomst verder kunnen worden onderzocht.

Hoofdstuk 3

Methodiek van de testen

Dit hoofdstuk behandelt de wijze waarop de testen naar consistentie en beschikbaarheid worden uitgevoerd. De methodiek is opgedeeld in 4 grote stappen: het opstellen, kalibreren, testen van de systemen en tenslotte het verzamelen en analyseren van de resultaten. Een overzicht van de procedure kan gevonden worden in figuur 3.1.

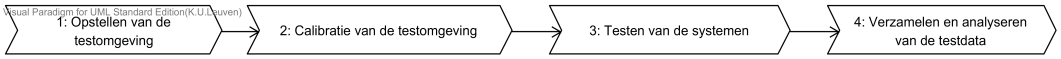
Opstellen van de testomgeving Deze eerste stap is voor het installeren en configureren van de DBMS en de testsoftware. Een variatie in hardware van de systemen, versienummer van de software of een verschillende netwerkinfrastructuur kan de uiteindelijke testresultaten beïnvloeden.

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

Testen van de systemen In deze stap worden de testen op de verschillende systemen uitgevoerd. Voor deze methodiek is het mogelijk om te testen hoe de vertraging op een bewerken zich gedraagt voor, tijdens en na het falen en herstellen van een systeem. Daarnaast is er ook een testmethode voor een actieve analyse van eventuele consistentie.

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

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



Figuur 3.1: Overzicht testproces

3.1 Stap 1: Opstellen van de testomgeving

Het lokaal installeren en configureren 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. Naast de lokale installatie en configuratie, is er ook een gedistribueerde configuratie stap.

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

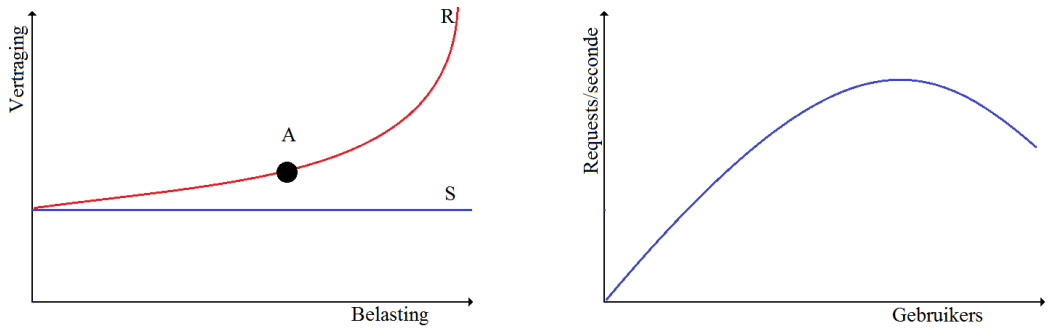
Configuratie bestanden Met deze methode dient er op elke lokaal systeem een configuratiebestand aangemaakt of aangepast worden met hierin een link naar één of meerdere andere instanties. Nadien worden de verschillende lokale systemen opgestart of de configuratiebestanden opnieuw ingeladen in de al draaiende instanties. Vervolgens zullen deze met de configuratie elkaar vinden en samen het database systeem vormen. Deze informatie kan een ip adres zijn van één of meerdere systemen maar dit kan ook een naam zijn van het systeem die met een broadcast verdeeld wordt.

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

Deze stap is gelijk aan het opzetten van het systeem in een productie omgeving; na het uitvoeren van deze stap, zou het DBMS volledig moeten werken.

3.2 Stap 2: Calibratie van de testomgeving

Afhankelijk van de onderliggende infrastructuur en het soort DBMS, kan het systeem een verschillend gedrag hebben onder dezelfde configuratie. Voor de eigenlijk testen is het de bedoeling om een middelmatige belasting te hebben. De queueing theorie geeft de eigenschap dat $R = S + W$ waar R de totale vertraging is, S de processing time en W de tijd in de wachtrij [23]. Dit verband is visueel voorgesteld ten opzichte van de belasting in figuur 3.2(a).



(a) Verband vertraging ten opzicht van de belasting van een DBMS.

(b) Verband aantal requests/seconde ten opzicht van het aantal gebruikers.

Figuur 3.2: Verbanden voor de calibratie

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

Hoeveelheid data per gegevensrecord Elke record in de database kan bestaan uit verschillende kolommen en per kolom een waarde. Het is belangrijk om te definiëren hoe groot een gemiddeld record is, aangezien dit een invloed heeft op het schijfgebruik en het netwerkverkeer.

Type van queries De opgeslagen data kan opgevraagd worden op verschillende wijze: data kan ingevoegd, aangepast, opgevraagd of verwijderd worden. Daarnaast kan dit gebeuren voor 1 of meerdere records tegelijk. Afhankelijk van de relatieve verhouding van deze soorten, kan een ander resultaat bekomen worden: sommige DBMS's zijn meer geschikt voor een dominantie in leesacties en vice versa.

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

Aantal connecties of gebruikers In een gedistribueerde omgeving zullen meestal 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: $\#Queries = \#Gebruikers * \#QueriesPerGebruiker$. In deze stap wordt er verondersteld dat de gebruiker het maximaal aantal queries doet, dus $1/Vertraging$. Rekening houdend met de exponentiële groei van de wachtrij vertraging (figuur 3.2(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, zie figuur 3.2(b).

Aantal queries per seconde In de vorige stap is er de optimale configuratie bepaald om het systeem maximaal te belasten. Maar in het begin is er gesteld dat er gezocht wordt naar een gemiddelde belasting voor dit aantal gebruikers. Er wordt gekozen om matige belasting, in figuur 3.2(a) zou dit punt A zijn.

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

3.3 Stap 3: Testen van de systemen

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

Opstellen van de database In stap 2 was er gekozen voor een bepaalde datastructuur, deze structuur wordt zo goed mogelijk meegegeven aan de DBMS zodat deze optimale allocatie kan doen.

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

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

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

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

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

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

3.3.1 Beschikbaarheidstest

Bij de beschikbaarheidstest wordt er gekeken hoe het systeem reageert op tijdelijk (on)verachte onbeschikbaarheid van een deel van het systeem. In deze testen worden er 3 mogelijke manieren getest die de systemen onbeschikbaar maakt, terwijl er de belasting uit stap 2 wordt toepast.

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

Harde stop De DMBS service wordt onmiddellijk gestopt door het process te beëindigen. De service heeft geen tijd om de andere te waarschuwen. Achteraf wordt dezelfde service terug opgestart. Dit simuleert een crash van de service die pas na enige tijd opgemerkt wordt.

Netwerk onderbreken Al het netwerk verkeer wordt gedropt 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 onbereikbare server om eender welke andere reden.

Een zelfde systeem kan sterk verschillend reageren op de verschillende situaties: waar de eerste situatie nog eenvoudig is te behandelen doordat het systeem de andere op de hoogte kan brengen, is de tweede situatie al moeilijker alhoewel andere systemen wel antwoord krijgt bij het contacteren dat de service niet beschikbaar is. De derde situatie is het moeilijkste te behandelen omdat men niet weet of de berichten naar de server niet aankomen of de antwoorden verloren gaan.

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

3.3.2 Consistentie test

In de consistentie test wordt onderzocht welke consistentie het DBMS ondersteund. Zoals voordien besproken in deel 2.2.2, bestaan er verschillende soorten.

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

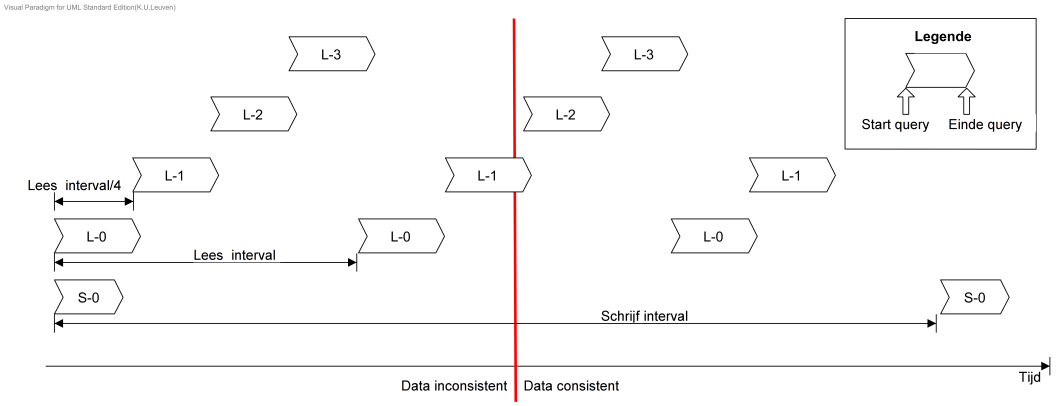
Beschrijving van de test Deze test bestaat uit 3 soorten gebruikers: er is 1 gebruiker die data schrijft (=S), een aantal lezer (=L's) en tenslotte zijn er nog andere gebruikers die zorgen voor de basisbelasting. De berekening van deze basisbelasting komt verder aanbod. Het is belangrijk dat er een exacte synchronisatie in tijd is tussen de verschillende gebruikers, dit om de geregistreerde tijdstippen te kunnen vergelijken.

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

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

Het plannen van de lezers Zoals voordien vermeldt, gaat de schrijver op bepaalde momenten schrijven en de lezer herhaalt het lezen periodiek. Het doel van de verschillende lezers is om allereerst verbonden te zijn met verschillende servers zoals ook gebruikers zullen zijn en daarnaast meer testpogingen hebben op het lezen van de data. Om deze laatste redenen worden de starttijdstippen voor de data te lezen, gelijk gespreid tussen de verschillende lezers. Een voorbeeldperiode met 1 schrijver en 4 lezers kan gevonden worden in figuur 3.3.

Schatten van de basisbelasting De basisbelasting kan de berekende belasting zijn in stap 2, waardoor het reëel aantal queries hoger ligt. De belasting kan ook verminderd worden met een geschat aantal queries die de schrijvers en lezers zullen uitvoeren. Het aantal queries van de schrijver en lezers per seconde kan berekend worden aan de hand van de hand van volgende formule: $(S + \#L * \#queriesperschrijfperiode/schrijfinterval$. Het aantal lees queries per



Figuur 3.3: Testen: Consistentie test met één periode. 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.

schrijf periode zal geschat moeten worden, maar kan bijvoorbeeld op 1 gezet worden. Op deze manier krijgen systemen die geen strikte consistentie afdwingen een hogere belasting om de correcte waarde te lezen.

Soorten eventuele consistentie Met deze uitgevoerde testen en data kan aangetoond worden dat bepaalde systemen bepaalde eventuele consistentie vereisten niet volgen. Het is in veel gevallen niet mogelijk om te bewijzen dat deze het wel uitvoeren omdat een voorbeeld niet sluitend is, maar een tegenvoorbeeld wel.

Strikte consistentie Een systeem is niet strikt 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.*

Check strikte consistentie!

Read your own writes consistentie Deze eventuele consistentie kan ontkracht worden indien een schrijver onmiddellijk na het voltooien zijn eigen data opvraagt en niet de nieuwe waarde leest. Dit kan enkel getest worden indien de DBMS het mogelijk maakt om met een gebruiker te verbinden naar meerdere servers.

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 een zelfde sessie. Dit kan ontkracht worden door met dezelfde connectie als de schrijver te lezen en nog de oude data te lezen.

Casual consistentie Deze test kan uitgevoerd worden de schrijver verschillende schrijfacties na elkaar te laten uitvoeren met elke schrijfactie onmiddellijk te lezen. De lezer leest de records in dezelfde of omgekeerde volgorde. Indien deze data van een latere schrijfactie leest maar nog niet van een vroegere, is dit ongeldig. De eis

kan strenger gemaakt worden door de schrijver tussendoor niet te laten lezen, dit zou andere resultaten kunnen hebben. Deze consistentie is in zijn totaal niet getest.

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

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

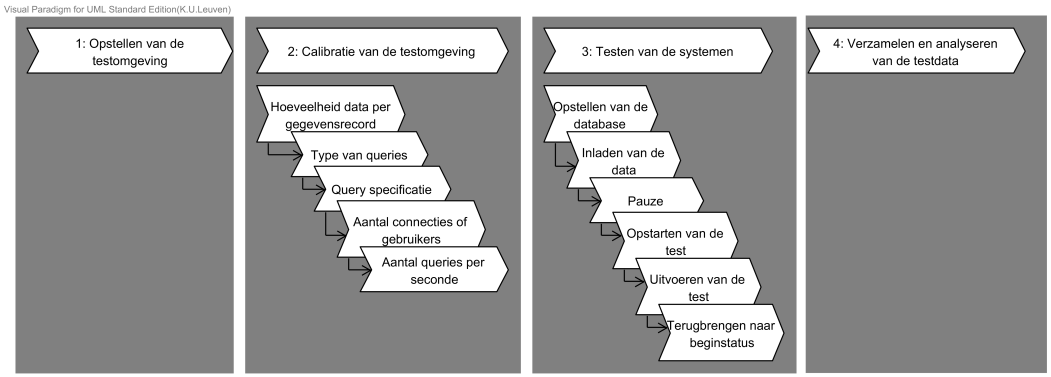
3.4 Stap 4: Verzamelen en analyseren van de testdata

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 overal consistent is (actieve analyse) of om tegenvoorbeeld te zijn voor een bepaalde consistentie categorie (passieve analyse). De analyse kan gebeuren op basis van de besproken methodes hierboven.

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

3.5 Conclusie

Een overzicht van de test methode kan gevonden worden in figuur 3.4. Deze testmethode geeft de mogelijkheid om in verschillende database systemen zowel de beschikbaarheid als de consistentie te testen op een gelijkaardige manier.



Figuur 3.4: Overzicht testproces

Hoofdstuk 4

Implementatie

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

Vooraleer met de eigenlijke uitwerking kon gebeuren, wordt eerst uitgelegd welke systemen gekozen zijn en waarom. Vervolgens zullen de gekozen systemen in meer detail besproken worden, waarna de uitwerking van de testsoftware aanbod komt. Tenslotte zullen de verschillende stappen van de testprocedure overlopen worden met hun gedetailleerde parameters.

4.1 Selectie van de DBMS's

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

Vrije software Om testen tussen verschillende DBMS's te kunnen vergelijken op een gelijkaardige infrastructuur, is het nodig dat deze software kan geïnstalleerd worden op de eigen infrastructuur, een extra selectie criteria is dat de systemen gratis aangeboden worden.

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

Replicatie Eén van de testen is de beschikbaarheidstest, indien de data maar op een enkele server opgeslagen is, zal de data op de uitgeschakelde server niet langer beschikbaar zijn. Met replicatie zal de data op verschillende servers opgeslagen

worden en zal de data in theorie nog beschikbaar zijn in het geval van een enkele uitgeschakelde server.

Data distributie Het is de bedoeling om systemen te testen die een grote hoeveelheid data kunnen opslaan. Om aan deze vereiste te voldoen, is het nodig dat elke server niet al de data opslaat bij een grote dataset. Hiervoor zijn er wel voldoende aantal servers nodig, bij te weinig servers is elke server nodig om aan de replicatie vereiste te voldoen.

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

Check table :-) en tekst hieronder

Voor alle systemen besproken in sectie 2.3, is het eerste criterium voldaan. De overige 4 criteria zijn samengevat in tabel 4.1.

Een korte verklaring bij enkele van de waarden uit de tabel. 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.

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. In deze thesis zijn HBase, MongoDB en Pgpool-II verder onderzocht, in de thesis van mijn collega zijn dit Cassandra, Apache CouchDB, Riak en MySQL.

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

Tabel 4.1: Ondersteuning van de besproken DBMS's naar de selectie criteria.

4.2 Gedetailleerde bespreking van de geselecteerde DBMS's

In dit gedeelte zal elk geselecteerd systemen in meer detail uitgelegd worden. Een gemeenschappelijk element bij al deze systemen is dat niet alle instanties dezelfde functie hebben (Master - Slave systemen), in andere DBMS's hebben allen dezelfde functie bij het wat de installatie kan vereenvoudigen (Master-Master).

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

4.2.1 HBase

Data structuur[12]

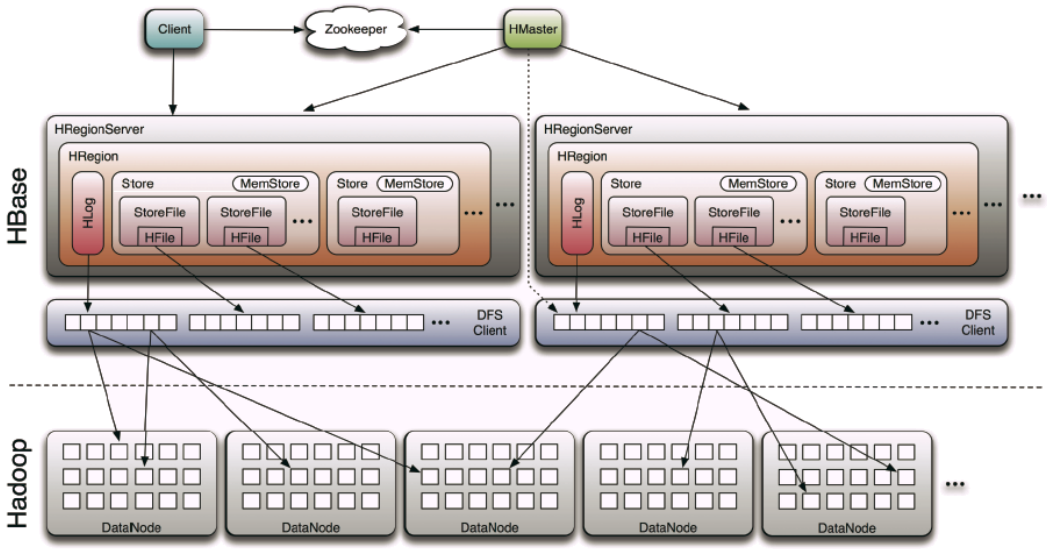
De data in HBase is gestructureerd in tabellen, bij het aanmaken er een schema voor de tabel gemaakt. Voor elke tabel kunnen de verschillende kolommen meegegeven worden samen met een *kolom familie* voor elke kolom, maar de kolommen kunnen ook gespecificeerd worden bij het schrijven van data. De gegevens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

De operaties beschikbaar in dit systeem zijn: get (verkrijgen), put (invoegen), scan en delete (verwijderen). Het aanpassen van gegevens wordt uitgevoerd via een put waarbij een enkele kolom waarde van een record kan aanpast worden. Een scan operatie heeft geen optie om het aantal op te halen records te bepalen maar er kan wel bepaald worden in welke batch grootte (bytes) de records opgehaald moeten

worden. Doordat er geweten is hoe groot een individueel record is én hoeveel records er opgevraagd worden, kan de cache grootte zo bepaald worden dat er maar een enkele datacommunicatie nodig is.

Architectuur[12]

De gedistribueerde versie van HBase is afhankelijk van 2 andere software systemen, namelijk Zookeeper[18] en Hadoop[3], en volgt hiermee de structuur van Google's BigTable[6] die op zijn beurt afhankelijk is van Chubby[5] en Google File System[13]. Een overzicht van de architectuur bevindt zich in figuur 4.1. De 3 systemen van HBase zullen kort besproken worden, van HBase naar Zookeeper en Hadoop.



Figuur 4.1: Volledige systeemarchitectuur van HBase met Hadoop en Zookeeper. Bron [19]

HBase[12] 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 van de HRegionServers in het oog. Daarnaast is deze ook verantwoordelijk voor het toewijzen van data verantwoordelijkheden, zoals het opsplitsen een tabel over verschillende regio's indien een tabel groeit en het toewijzen van een regio aan een HRegionServer.
De andere soort, **HRegionServer**s, is verantwoordelijke voor de data in en voor het beheren van regio's. Een regio is een deel van een tabel met daarin de feitelijke data die opgeslagen is in verschillende datanodes. Een HRegionServer zal strikte consistentie afdwingen in HBase op een enkele record.

Hadoop[3] HBase maakt gebruik van het Hadoop Distributed File System (HDFS), een gedistribueerd file systeem ontworpen om te werken op commodity hardware met een hoge fout tolerantie. HDFS heeft een master/slave architectuur en bestaat

uit een enkele **namenode**, de master server, die de naamruimte en toegangscontrole onderhoudt, en **datanode**s. De data wordt opgedeeld in blokken die door een verzameling van datanodes wordt opgeslagen, op deze manier is er data distributie. Deze master/slave configuratie zijn verschillende soorten van services en dient door de gebruiker zelf geconfigureerd te worden.
In de deze configuratie van HBase, is HDFS de methode om data persistent op te slaan met automatische replicatie en data distributie. Er is ook ondersteuning om de opslag naar Amazon S3 te doen in een gedistribueerde omgeving of deze op de lokale harde schrijf op te slaan bij een configuratie met slechts 1 server.[12]

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 op zijn beurt een gedistribueerd master/slave systeem dat ontworpen is om snel te zijn bij dominantie van leesoperaties.
HBase gebruikt Zookeeper onder andere voor het bijhouden van de status van regio server, hun locatie en hun verantwoordelijkheden. Dit verloopt met het toekennen van sessie die een een HRegionServer bijvoorbeeld de verantwoordelijkheid voor een Region geeft voor de volgende minuut. Tijdens deze periode kan geen enkele andere HRegionServer een bewerking doen op deze Region, uitgezonderd met de toestemming van de verantwoordelijke server. [12]

Uitleggen van ticktime

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 HDBS namenode, een Zookeeper instantie en de HMaster worden samen op een enkele instantie geplaatst, hetzelfde geldt voor een datanode en een HRegionServer. Zeker deze laatste heeft een extra performantie invloed: HBase detecteert dat er lokale opslag van de data is en de regio zal steeds deze lokale opslag hebben. Dit zorgt bij leesacties voor een performantie verbetering aangezien de data lokaal gelezen kan worden.

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

4.2.2 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 een database en collectie op voorhand aan te maken, deze worden automatisch aangemaakt bij het wegschrijven van data indien de collectie nog niet bestaat. Een record is in MongoDB een document en elk record kan verschillende velden hebben. Er zijn uitgebreide query mogelijkheden om data

in te voegen, aan te passen, te verwijderen of een scan uit te voeren. Er is ook ondersteuning voor MapReduce[10].

Bij het schrijven van data, kunnen verschillende eisen gesteld worden voor het voltooien van de actie, startende met de actie is over het netwerk verstuurd, de primary heeft de data geschreven tot een meerderheid van de secundaries heeft de data weg geschreven.
Bij het lezen kan men kiezen om de data te lezen van de primary, secundary of de dichtstbijzijnde node. Afhankelijk van de gekozen acties, kan er verondersteld worden dat er een verschillende consistentie garantie zal zijn. Een overzicht van al de mogelijkheden, kan teruggevonden worden in tabel 4.2.

Lees acties

Benaming	**Omschrijving**
Primary	Enkel lezen van de primary
PrimaryPreferred	Lezen van de primary, behalve als de primary onbeschikbaar is, lees dan van secundary.
Secundary	Enkel lezen van een secundary
SecundaryPreferred	Lezen van een secundary, behalve als er geen secundary onbeschikbaar is, lees dan van de primary.
Nearest	Lees van de instantie met de laagste netwerk vertraging, ongeachte het een primary of secundary is.

Schrijf acties

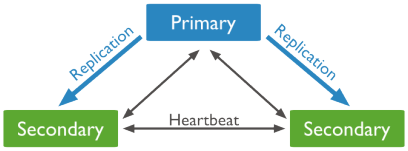
Benaming	**Omschrijving**
Normal	Wacht tot weggeschreven naar het netwerk socket.
Safe	Wacht op bevestiging van de primary
fsync_safe	Wacht op bevestiging van de primary tot de data is weggeschreven naar harde schijf.
Replica acknowledged	Wacht op bevestiging van primary en één secundary.
Majority	Wacht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

Tabel 4.2: MongoDB: Mogelijke opties bij lees- en schrijfqueries

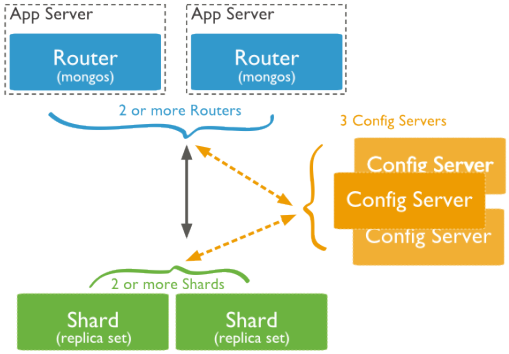
Architectuur

MongoDB is een DBMS dat de vereisten van replicatie en data distributie op een gelaagde manier tot uitvoering brengt. In eerste instantie zal deze de replicatie vereisten invullen, hierboven zal horizontale schaalbaarheid ondersteund worden.

Replicatie[26] Replicatie in MongoDB gebeurt door middel van een master/slave configuratie tussen verschillende **MongoD** instanties, of in hun termen primary/secundaries. Deze instanties verkiezen zelf hun primary die verantwoordelijk is voor het afhandelen van de schrijfacties, de data zal vervolgens gerepliceerd worden naar



(a) Drie leden van een replica set met een primary en 2 secundaries.



(b) Een voorbeeld cluster voor productie met 2 mongos, 3 shards en 3 configuratie servers.

Figuur 4.2: MongoDB Architectuur voor replicatie en datadistributie. Bron figuur 4.2(a): [26], figuur 4.2(b): [27]

de secundaries, of dit synchroon of asynchroon gebeurt is afhankelijk van de optie bij de schrijfactie. Een verzameling van deze MongoD instanties wordt een *replicaset* genoemd. 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 een enkele MongoD instantie te combineren tot een cluster. In het geval van de tweede keuze, zal de data niet gerepliceerd worden en wordt om deze reden niet aangeraden voor productie.

Shards Sharding gebeurt automatisch op een collectie nadat is aangegeven dat men deze wilt verdelen over de gespecificeerde delen. Voor het uitvoeren van deze sharding zijn er nog 2 extra servers types nodig: configuratie servers en toegangsserver.

Configuratie servers De configuratie servers slaan de meta data van de cluster op zoals de verschillende shards en replicaset's. Deze configuratie set bestaat in productie uit exact 3 servers maar kan ook bestaan uit een enkele configuratie server.

Toegangsserver De toegangsserver haalt de configuratie op uit de configuratie servers en biedt toegang voor de gebruiker aan tot de cluster. Er kunnen een onbepaald aantal toegangsservers zijn in cluster.

De configuratie van de verschillende delen gebeurt op verschillende manieren. 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.

4.2.3 Pgpool-II (PostgreSQL)[29]

Pgpool-II kan op 4 verschillende manieren werken, in deze testen is er gekozen voor de replicatie mode omdat deze zowel replicatie, belastingsverdeling, failover en online recovery aanbiedt. Er is de mogelijkheid om ook data distributie aan te bieden maar dit is niet getest. Beide kunnen gecombineerd worden door de datadistributie voor de replicatie te zetten, hetzelfde principe als MongoDB.

Datastructuur en de architectuur van Pgpool-II in parallelle mode komt nu in meer detail aanbod.

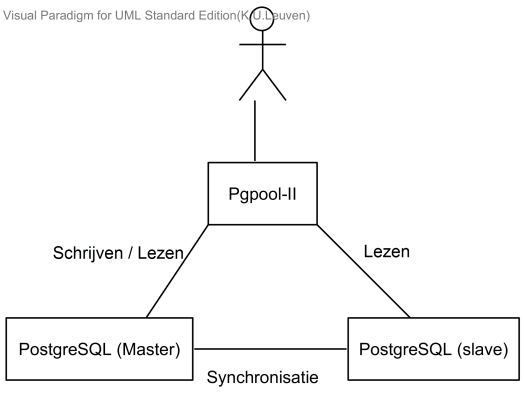
Datastructuur

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. In een database zijn vervolgens verschillende tabellen die de records bevatten. Voor het opslaan van de data dient de volledige tabel met al de kolommen gespecificeerd zijn.

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

Architectuur

Een Pgpool-II infrastructuur bestaat uit 2 delen, een data en routing niveau, een overzicht is gegeven in figuur 4.3.



Figuur 4.3: Systeemarchitectuur van Pgpool-II.

Op data niveau bestaat een individuele service uit een PostgreSQL installatie waarbij extra functies en bestanden worden geïnstalleerd met een aanpassing aan de configuratie bestanden. Daarnaast moet er voor de online recovery ook ssh toegang voorzien worden tot al de PostgreSQL servers. De verschillende data machines hebben een master/slave structuur waar al de schrijfacties naar de master worden gestuurd en de leesoperaties zijn verdeeld over al de machines. De master doet aan synchronisatie met behulp van de *Write-ahead-log* van PostgreSQL waar al de schrijfactie worden gelogd.

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. Bij het aanmaken van een database connectie naar welke service de leesoperaties zullen gaan, zo wordt de leesbelasting verdeeld.

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

4.3 Selectie en uitwerking van de testsoftware

De testen zijn geïmplementeerd als een uitbreiding van YCSB[9] omwille van verschillende redenen. Allereerst is de broncode publiek beschikbaar onder Apache 2.0, daarnaast is dit een uitgebreid systeem voor het uitvoeren van performantie benchmarking, dit op basis van het meten van de vertraging op een query voor verschillende DBMS's. Hierdoor heeft deze al een uitgebreide ondersteuning voor tal van DBMS's, waaronder al de gekozen systemen. Wel is deze ondersteuning nog verder geoptimaliseerd voor de gekozen systemen zodat er maximaal gebruikt wordt van de functionaliteiten van het systeem. Een concreet voorbeeld, bij het opstellen van de scan queries rekening gehouden wordt met het aantal records dat nodig is wat standaard in YCSB niet gebeurd bij het uitvoeren op een relationele database.

De 2 testen, beschikbaarheidstest en consistentie test, worden op verschillende manieren geïmplementeerd.

Beschikbaarheidstest De beschikbaarheidstest wordt geïmplementeerd door middel van *event support*, hiermee kan er op vooraf gedefinieerde momenten een bepaald Unix commando uitgevoerd worden. De configuratie gebeurt met behulp van een XML bestand met de parameters van 4.3 die meegegeven wordt aan de parameter $eventFile$, de output komt in het logbestand met de elementen van tabel 4.4.

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

Naam	eenheid
ID	String
Starttijdstip	milliseconden
Commando	String

Tabel 4.3: Configuratie van event support

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

Tabel 4.4: Uitvoer van event support

Consistentie testen Voor de consistentie testen is er een extra module geïmplementeerd die dit gedrag uitvoert. In deze uitwerking leest de schrijver niet zijn eigen data, al zou dit eenvoudig mee geïmplementeerd kunnen worden, dit is niet getest omdat het niet nodig was in deze testen. De testen kunnen uitgebreid geconfigureerd worden om enkel te testen wat nodig is: een overzicht van de configuratie parameters, uitgezonderd de locatie van de logbestanden, is te vinden in tabel 4.5. Voor elke uitgevoerde query, wordt een record aangemaakt met de data van tabel 4.6.

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-ConsistencyCheck	Float $(0 \leq x \leq 1)$	Het percentage van schrijfacties dat een nieuw record invoegt
updateProportion-ConsistencyCheck	Float $(0 \leq x \leq 1)$	Het percentage van schrijfacties dat een record aanpast
stopOnFirstConsistency	Boolean	Stop zodra de eerste keer een correct record is gelezen
maxDelayConsistency-BeforeDropInMicros	Micro-seconden	De maximale afwijking tussen de eigenlijke start van de query en het geplande moment
timeoutConsistency-BeforeDropInMicro	Micro-seconden	De maximale tijd dat een leesactie geprobeerd wordt

Tabel 4.5: Configuratie van de consistentie testen

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

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 geduurd
Waarde	String	De gelezen of geschreven waarde

Tabel 4.6: Uitvoer van een enkel query in de consistentie testen

4.4 Installatie en opstelling van de DBMS's en YCSB

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

De automatisatie gebeurt met het Integrated configuration Management Platform (IMP) beschreven in [39]. Dit modulair framework is uitgebreid met de 3 DBMS's en YCSB! waardoor de configuratie als een declaratief gewenste staat wordt uitgedrukt. IMP zal deze staat toepassen op de verschillende systemen bij het uitrollen.

Een uitgebreider bespreking van de uitwerking in IMP kan gevonden worden in appendix A met het domeindiagram, uitleg en voorbeeldcode.

Voor de uitvoering van de testen, is er voor elk DBMS gekozen voor een minimaal aantal instantie dat datadistributie én replicatie ondersteunt, voor de laatste eigenschap zou de data beschikbaar moeten zijn bij het uitvallen van 1 server. In de testen is er enkel gefocust op het uitvallen van dataservers, niet naar configuratieservers. Om deze reden zijn configuratie en toegangsservers minimaal opgezet.

De opstelling van de systemen is getoond in figuur 4.4, elk van de systemen zal in meer detail besproken worden nadat de testinfrastructuur is besproken.

De testinfrastructuur is een IaaS (Infrastructure as a Service) gebaseerd op OpenStack[1]. De infrastructuur bestaat uit 3 Dell R610 en R620 servers met een totaal van 196GB RAM, 44 fysische CPU's (88 met hypertreading), verbonden met een Gigabit switch. Deze infrastructuur is gedeeld met andere gebruikers. Elke instantie heeft 2 virtuele CPU's, 4GB RAM en 50GB schijfruimte. De instanties worden verdeeld over de verschillende servers.

[1] https://www.openstack.org/

HBase Voor HBase wordt de data standaard 3 maal gerepliceerd en zijn er voor datadistributie dus 4 data instanties nodig die elk een HBaseRegionServer en Hadoop datanode zijn. Verder is er nog de nood aan een HMaster, Zookeeper en Hadoop namenode die samen op een enkele instantie worden uitgerold. Daarnaast zijn er nog 2 andere Zookeeper instanties en HMasters. Als *ticktime* in Zookeeper wordt 2s gekozen met *synchLimit* 5. In het totaal zijn er 7 instanties. Een overzicht van de infrastructuur getoond in figuur 4.4(a). De configuratiebestanden kunnen gevonden worden op `https://github.com/thuys/hbase` in de folder *templates*.

check

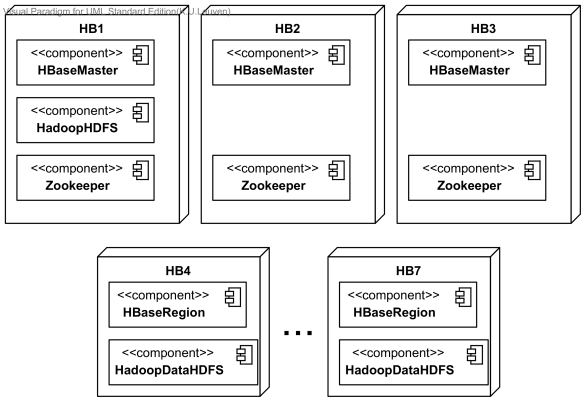
Pgpool-II Bij Pgpool-II is er ondersteuning voor horizontale schaalbaarheid in de parallel mode maar dit is niet getest. Om deze reden is er enkel replicatie toegepast waarvoor er 3 instanties zijn: een Pgpool-II instantie en twee PostgreSQL instanties. De configuratie van deze instanties zijn standaard 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 4.4(b). De configuratiebestanden kunnen gevonden worden op `https://github.com/thuys/postgresql` in de folder *templates*.

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 identificatie. Omdat de toegangsserver en configuratie instanties niet veel resources innemen, zijn deze verspreid over de verschillende data instanties, er zijn meerdere toegangsnodes geplaatst om bij de beschikbaarheidstesten steeds aan één node te kunnen. Zo zijn er in totaal 6 instanties nodig. Deze zijn beschreven in 4.4(c). De configuratiebestanden kunnen gevonden worden op `https://github.com/thuys/mongodb` in de folder *templates*.

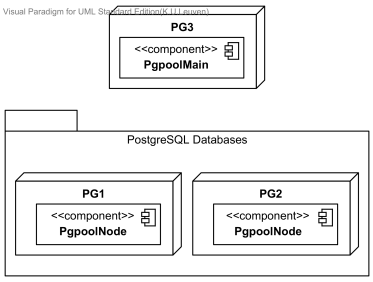
YCSB! YCSB! kan naar meerdere instanties uitgerold worden. Bij de calibratie-testen zullen er 2 instanties gebruikt worden, maar er zal blijken dat dit niet een limiterende factor is. Voor het uitvoeren van de eigenlijke testen, zal er gebruik gemaakt worden van een enkele YCSB! instantie, namelijk VM-1. Een overzicht is getoond in figuur 4.4(d).

4.5 Uitvoeren van de calibratie en testen

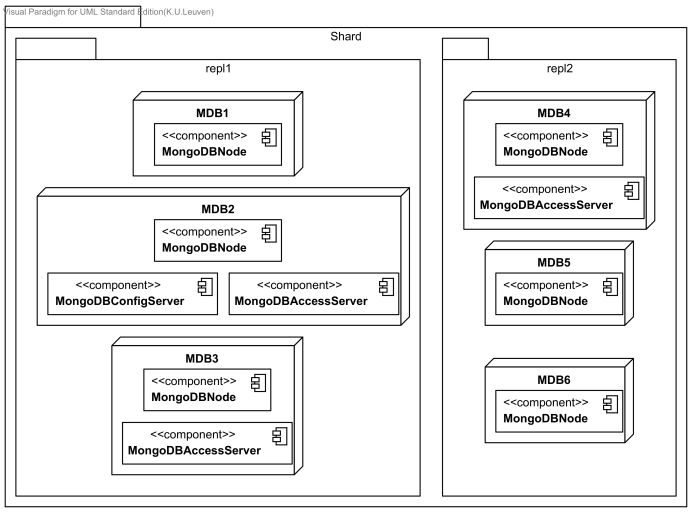
Voor het uitvoeren van de volledige benchmarking dient eerst de verdeling van de type queries gespecificeerd worden, deze zijn voor alle verschillende systemen gelijk. Een overzicht van deze parameters kunnen gevonden worden in tabel 4.7. 40% van de uitgevoerde queries past de database aan, er is dus een dynamische database. Bij het lezen wordt er de helft van de keren in batch gelezen met gemiddeld gezien 50 records per keer. Er wordt zo veel sequentiële records gelezen. Tenslotte wordt er



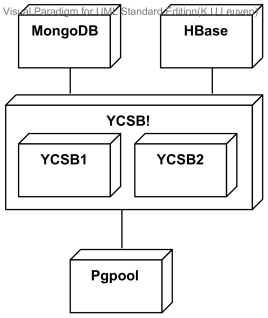
(a) Deployment van HBase met 7 instanties.



(b) Deployment van Pgpool-II met 3 instanties.



(c) Deployment van MongoDB met 6 instanties.



(d) Deployment van de testomgeving met 2 YCSB instanties.

Figuur 4.4: Deployment van de verschillende DBMS's en de testomgeving.

met *zipfian* gekozen om regelmatig dezelfde records te lezen waardoor er uit cache gelezen worden.

Calibratie testen Voor de calibratie van de omgeving zijn er 2 soorten testen gedraaid, de parameters voor het aantal connecties kunnen gevonden worden in tabel 4.8. De parameters voor het aantal queries per second zijn te vinden in tabel 4.9, in dit geval is het aantal gebruikers afhankelijk van de vorige test.

Check of tabel nog correct

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

Tabel 4.7: 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 4.8: Calibratie: Overzicht van de parameters voor het testen van het aantal gebruikers

Naam	**Waarde**	
	HBase en MongoDB	**Pgpool-II**
Ingeladen records	300 000	300 000
Pauze	50s	50s
Executie tijd	600s	600s
Theoretisch aantal records per seconde	50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1500, 2000, 3000, 4000	20, 50, 100, 150, 200, 250, 300, 400, 500, 600, 700, 800, 900, 1000

Tabel 4.9: Calibratie: Overzicht van de parameters voor het testen van het aantal records per seconde

Beschikbaarheidstesten Bij het uitvoeren van de testen op beschikbaarheid van de verschillende systemen zijn de parameters in tabel 4.10 gebruikt. De commando's voor het stoppen en starten van de systemen zijn te vinden in tabel 4.11. Voor Pgpool-II is er een extra commando toegevoegd dat na het herstarten van de systemen wordt uitgevoerd, dit komt omdat er geen automatische recovery in Pgpool-II is. Tenslotte worden deze testen uitgevoerd op al de datanodes, een overzicht hiervan met de overeenkomstige service is te vinden in tabel 4.12.

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

Tabel 4.10: Beschikbaarheidstesten: Overzicht van de parameters

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

Naam	Instanties	Service naam
HBase	HB3, HB4, HB5, HB6	hbase-regionserver hadoop-hdfs-datanode
MongoDB	MDB1, MDB2, MDB3,	mongodb-dataserver
Pgpool-II	PG1, PG2	postgresql

Tabel 4.12: Beschikbaarheidstesten: Overzicht van de instanties naar figuur 4.4

Check of het er nu echt 6 zijn

Consistentie testen Voor de consistentie testen moeten de parameters van tabel 4.5 geconfigureerd worden, de parameters zijn te vinden in tabel 4.13. Deze test wordt uitgevoerd op HBase en MongoDB, om de analyse van de gegevens eenvoudiger te maken is er bij MongoDB gekozen om de test enkel uit te voeren op een replicaset en niet op een volledige cluster. Er is een aanname gedaan dat het consistentie venster bepaald is door de tijd dat het duurt dat de gegevens beschikbaar zijn op

Naam	Waarde
Ingeladen records	300 000
Pauze	50s
Executie tijd	900s
starttime	10s
readThreads	30
consistencyDelayMillis	60ms
newrequestperiodMillis	500ms
readProportionConsistencyCheck	50%
updateProportionConsistencyCheck	50%
stopOnFirstConsistency	True
maxDelayConsistencyBeforeDropInMicros	300ms
timeoutConsistencyBeforeDropInMicro	300ms

Tabel 4.13: Consistentie testen: Overzicht van de parameters

al de verschillende instanties van een replicaset, het testen van een cluster voegt zo extra complexiteit toe. Deze test zou in de toekomst ook uitgevoerd kunnen worden op een cluster maar is in dit geval niet gedaan.

4.6 Verzamelen en analyse van de testresultaten

Te schrijven

Hoofdstuk 5

Observaties

5.1 Calibratie

De resultaten van de calibratietest voor het aantal gebruikers kunnen gevonden worden in figuur 5.1. Het aantal gebruikers wordt gekozen als het eerste moment waarop het de totale doorvoer vermindert, dit zorgt voor de gegevens in tabel 5.1.

Check data in tabel

DBMS	Aantal gebruikers
HBase	40
MongoDB	20
Pgpool-II	30

Tabel 5.1: Calibratie: Aantal gebruikers per test voor de verschillende DBMS's

De resultaten voor de calibratietest voor het aantal requests per seconden kunnen gevonden worden in de figuren 5.2, 5.3 en 5.4 voor respectievelijk HBase, MongoDB en Pgpool-II. Aan de hand van deze data wordt een getal gekozen voor het aantal queries per seconde zodat er een matige belasting is. Dit zorgt voor de gegevens in tabel 5.2.

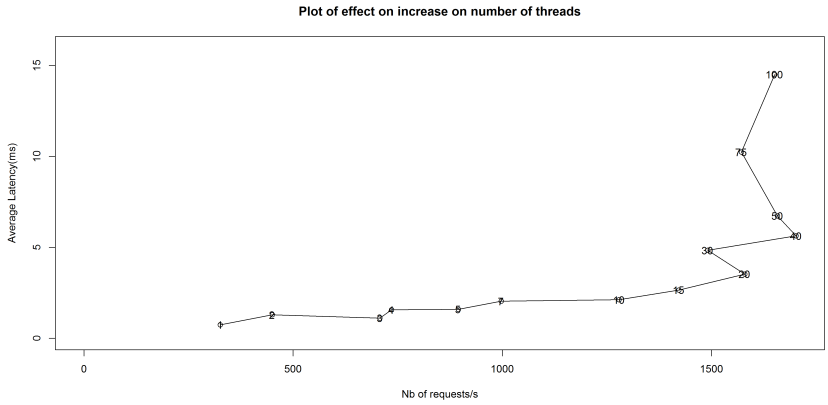
Check data in tabel

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

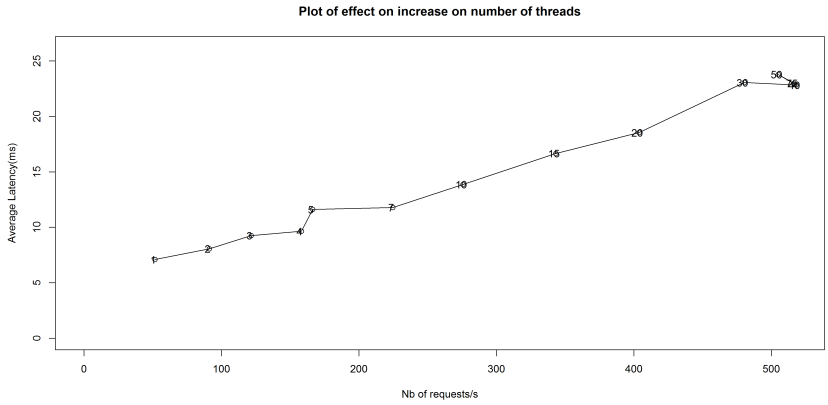
Tabel 5.2: Calibratie: Aantal queries per seconde per test bij een matige belasting voor de verschillende DBMS's.

5.2 Beschikbaarheidstest

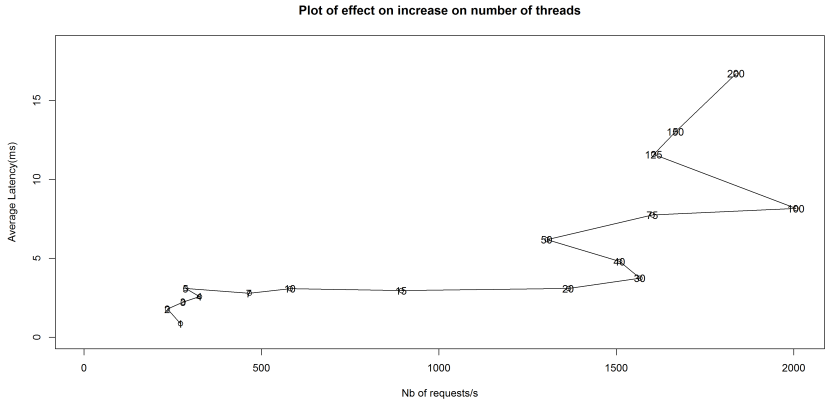
5.3 Consistentie test



(a) MongoDB

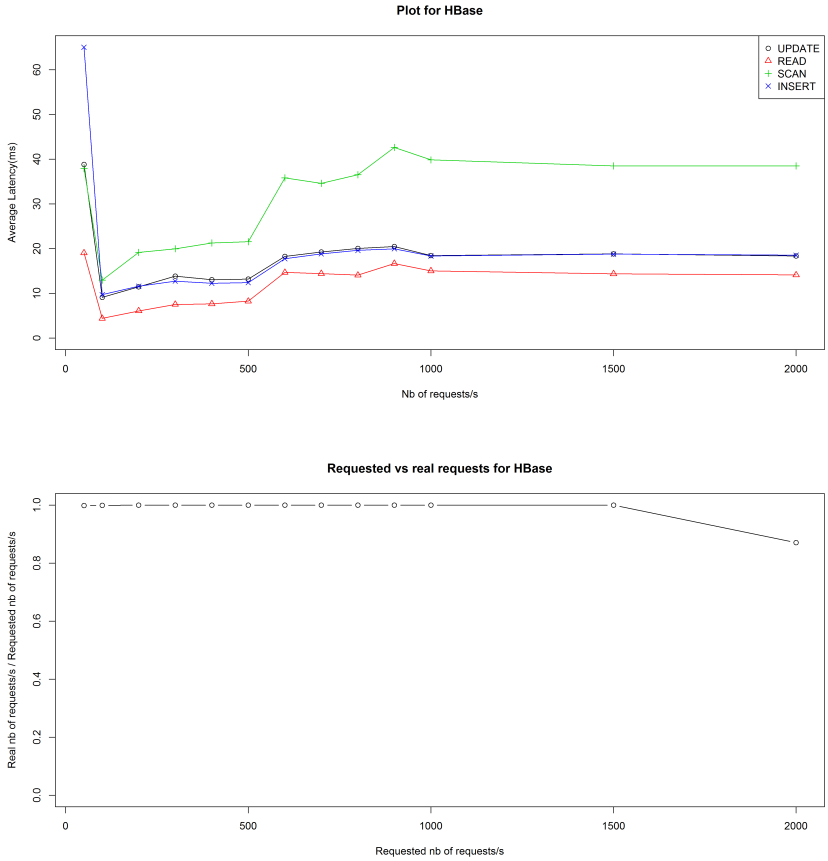


(b) Pgpool-II

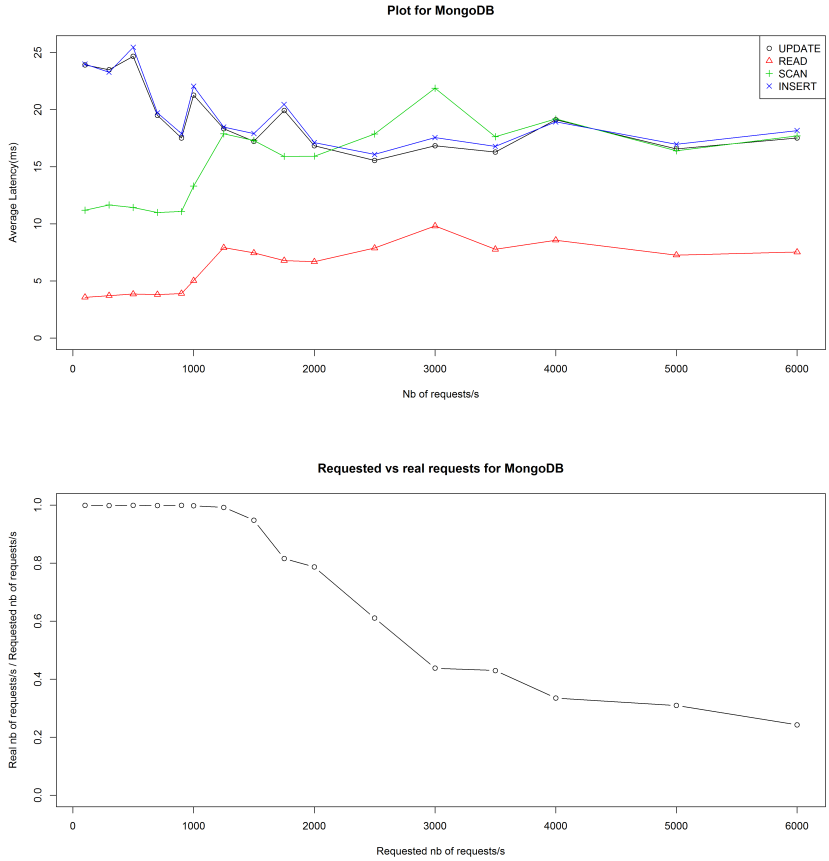


(c) HBase

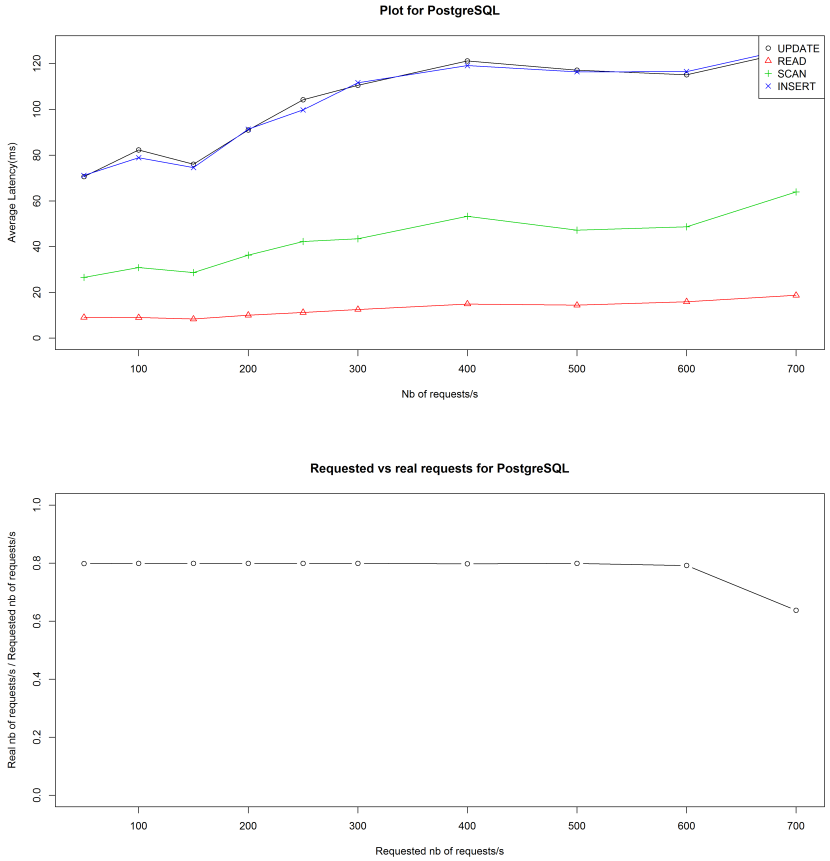
Figuur 5.1: Calibratie: Overzicht van het aantal requests tot de gemiddelde vertraging voor verschillend aantal gebruikers. Elk datapunt stelt een gebruiker voor met het aantal in het punt.



Figuur 5.2: Calibratie: Overzicht van de vertraging t.o.v. het theoretisch aantal aanvragen met een vergelijking hoeveel werkelijke aanvragen er waren voor HBase.



Figuur 5.3: Calibratie: Overzicht van de vertraging t.o.v. het theoretisch aantal aanvragen met een vergelijking hoeveel werkelijke aanvragen er waren voor MongoDB.



Figuur 5.4: Calibratie: Overzicht van de vertraging t.o.v. het theoretisch aantal aanvragen met een vergelijking hoeveel werkelijke aanvragen er waren voor Pgpool-II.

Hoofdstuk 6

Analyse van de resultaten

6.1 Calibratie

Over de calibratie testen valt in het algemeen niet veel af te leiden, deze testen zijn niet uitgevoerd op een volledig dezelfde infrastructuur zo heeft Pgpool-II slechts 3 instanties t.o.v. 6 voor HBase. Wel is er een interessante observatie bij het aantal queries per seconde.

Aantal queries per second Bij verdere analyse van de testresultaten van het aantal queries per seconde, blijkt dat elk van testen afvlakt door de beperkte netwerk bandbreedte, 100MB/s volgens iftop[1], en niet door een beperking van schrijftoegang of CPU.

Een berekening van de gemiddelde kost per query zoals getoond in tabel 6.1, kan deze afvlakking niet onmiddellijk verklaren. Met 2000 queries per seconde voor HBase, wordt er slecht 20MB/s aan data over het werk verstuurd. Maar in deze analyse is er enkel rekening gehouden met de data, noch met het resultaat van een query, noch met overhead van HBase en het netwerk protocol.

Check

Soort	**Uitgaand verkeer**	**Inkomend verkeer**
Invoeg queries (20%)	200 byte	
Lees queries (40%)		400 byte
Aanpas queries (20%)	200 byte	
Range queries (20%)		10 000 byte
Totaal	400 byte	10 400 byte

Tabel 6.1: Gemiddeld netwerk verkeer per query enkel voor het overbrengen van data

[1] Een bandbreedte monitoring tool

6.2 Beschikbaarheidstest

Bij de beschikbaarheidstesten lijkt er uit de resultaten dat de verschillende systemen een andere aanpak hebben genomen. Deze zullen nu verder in detail besproken worden. Belangrijk is dat er hier verschillende queries uitgevoerd worden, waardoor er data van de verschillende datadistributies gelezen zal worden.

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

De duur van een sessie kan geconfigureerd worden in Zookeeper met behulp *tickTime*

updten

HBase: Zachte stop Bij een zachte stop, is er slechts af en toe sprake dat dit merkbaar is, de verklaring hiervoor is dat dit enkel wordt opgemerkt als de RegionServer die op dat moment verantwoordelijke is voor de Region wordt stopgezet. Indien deze RegionServer wordt stopgezet, is er geen onderbreking in de queries maar nemen deze tijdelijk meer tijd in beslag, tot 600ms in plaats van gemiddeld genomen rond de 10ms. Het terug online brengen van de server heeft geen invloed op de snelheid een query wordt uitgevoerd . Na het stopzetten van de RegionServer is er een verhoogde vertraging in beide leesoperaties (range en lees).
Zodra er een herverdeling is van de Regions over de aanwezige Regionservers, verdwijnt deze verhoogde vertraging. Wel heeft dit als gevolg dat er tijdelijk opnieuw een verhoogde vertraging is gelijkend aan deze bij het zacht uitschakelen van een instantie, zoals te zien rond tijdstip 500.

HBase: Harde stop Bij een harde stop, is er opnieuw slechts in enkele gevallen sprake dat dit merkbaar is. Dit heeft ook hier te maken dat dit enkel wordt opgemerkt als de stopgezette RegionServer verantwoordelijk is voor een Region.
Op dat moment worden alle queries stopgezet tot de sessie van de RegionServer op die Region verlopen is en een nieuwe RegionServer is gekozen.

Hadoop: Zachte stop Bij het zacht stoppen van een Hadoop Datanode, is er tijdelijk een hogere vertraging in een query, van 10ms tot meer dan 100ms. Dit effect neemt binnen de seconde terug af waarna er geen effect meer zichtbaar is op de vertraging van de queries. Dit wordt veroorzaakt omdat bij het wegschrijven dit wordt geregistreerd en er een andere datanode moet gevonden worden.

Hadoop: Harde stop Het effect is gelijk aan deze van een zachte stop, het wegschrijven naar de een datanode lukt niet en er wordt dus een andere gezocht.

Netwerk onderbreking Bij een netwerk onderbreking, worden de queries tijdelijk stopgezet en in veel gevallen is er zelfs sprake van 2 keer een onderbreking. Deze onderbreking kan lange tijd duren en is opnieuw afhankelijk van Zookeeper.

MongoDB Bij MongoDB is er tussen de leden van een Replicaset een heartbeat protocol. Indien er gedurende 10 seconden geen antwoord op een een heartbeat komt, wordt een server als offline bestempelt. Dit heeft opnieuw zijn invloed op de verschillende soorten stopzetten.

Bij MongoDB is het zo dat de vertraging zeer onregelmatig is, dit heeft te maken met het gebruik van locking in MongoDB. Bij het parallel uitvoeren van verschillende lees- en schrijfacties, worden er locks gelegd op de volledige database. Door een reeks van lees queries, kunnen de schrijfacties tijdelijk uitgesteld worden, wat voor een grotere vertraging zorgt. [24].

Zachte stop Bij een zachte stop is er een kans van 1 op 3 dat het uitvallen van een instantie zichtbaar is, dit is te verklaren doordat enkel het uitschakelen van de primary een invloed zal hebben op de vertraging , in de standaard modus werd er enkel gelezen naar en geschreven van de primary. Nadien is er geen invloed bij de verschillende queries naar de vertraging.

vertraing toevoegen

Harde stop Bij een harde stop is er een kans van 1 op 3 dat het uitvallen van een instantie zichtbaar is, net zoals bij een zachte stop.

vertraing toevoegen

Netwerk onderbreking

Check met data

Pgpool-II Bij Pgpool-II wordt er bij het hebben van een connectie naar Pgpool-II, de connecties naar de verschillende PostgreSQL instanties gecontroleerd. 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 wel verschillende interactie reacties op de verschillende problemen.

Een vereiste bij het herstellen van een instantie is dat er op dat moment geen enkele gebruiker actief is.

Zachte stop Bij een zachte stop van een data instantie worden alle verbinden met Pgpool-II 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, bij een grote hoeveelheid data instanties zal dit effect kleiner worden. Zodra de recovery gestart wordt, zal deze eerst op de huidige master de data verzamelen en vervolgens dit doorsturen naar de te herstellen database. Om ervoor te zorgen dat de te herstellen database dezelfde data heeft,

wordt er hiervoor gewacht op een moment dat er geen connecties zijn. In de testen blijven er gebruikers actief waardoor het herstel niet lukt. Wel is effect van de poging tot herstel zichtbaar op de belasting van het systeem na tijdstip 600.

Harde stop Een harde stop reageert hetzelfde als een zachte stop, dit omdat ook in deze wijze de verbindingen worden gebroken.

Netwerk onderbreking Bij een netwerk onderbreking is er een ander gedrag, de queries wachten op een antwoord maar krijgen dit niet. Hierdoor wordt er gewacht op de time-out die hier

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

6.3 Consistentie test

HBase HBase garandeert strikte consistentie op een enkel record en hoe deze garantie tot uitvoering wordt gebracht, is duidelijk zichtbaar. Een lees query wordt namelijk op wacht gezet tot de schrijf query voltooid is. In figuur 6.1 wordt het lees- en schrijfmodel van HBase uitgelegd naar Lars Hofhansl[16]. Samen met het gebruik van sessies voor een bepaalde Region, is het eenvoudig om de locking te doen.

In enkele gevallen zal de oude waarde nog gelezen worden, op dit moment wordt het oude leespunt dus gelezen en is de data nog niet beschikbaar.

MongoDB MongoDB biedt strikte consistentie aan als er van de primary gelezen wordt. Maar er zijn ook andere schrijf methodes zoals voordien besproken.

Schrijven

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

Lezen

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

Figuur 6.1: HBase: Het vereenvoudigde lees- en schrijfmodel voor strikte consistentie in HBase naar Lars Hofhansl[16]

Hoofdstuk 7

Conclusie

Bijlagen

Bijlage A

Uitwerking IMP

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] E. F. Codd. „A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (jun 1970), p. 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685. URL: http://doi.acm.org/10.1145/362384.362685.

[8] Edgar F Codd. „A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6 (1970), p. 377–387.

[9] 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.

[10] Jeffrey Dean en Sanjay Ghemawat. „MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), p. 107–113.

[11] Ramez Elmasri en Shamkant Navathe. *Fundamentals of Database Systems*. 6th. USA: Addison-Wesley Publishing Company, 2010. ISBN: 0136086209, 9780136086208.

[12] Lars George. *HBase: the definitive guide*. "O'Reilly Media, Inc.", 2011.

[13] 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.

[14] Wojciech Golab e.a. „Eventually consistent: not what you were expecting?" In: *Communications of the ACM* 57.3 (2014), p. 38–44.

[15] Jim Gray. „Data Management: Past, Present, and Future". In: *arXiv preprint cs/0701156* (2007).

[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 10-07-2014).

[30] Pouria Pirzadeh, Junichi Tatemura 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 categoriza- tion*. 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] Tilmann Rabl e.a. „Solving big data challenges for enterprise application performance management". In: *Proceedings of the VLDB Endowment* 5.12 (2012), p. 1724–1735.

[35] Arnaud Schoonjans. „Een critische evaluatie van beschikbaarheid in gedistribueerde opslag systemen". KU Leuven, 2014.

[36] Ben Scofield. *NoSQL – Death to Relational Databases(?)* Jan 2010. URL: http://www.slideshare.net/bscofield/nosql-codemash-2010 (bezocht op 03-02-2014).

[37] Christof Strauch. *NoSQL Databases*. 2010. URL: http://www.christof-strauch.de/nosqldbs.pdf.

[38] 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.

[39] 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.

[40] 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: Automatisch uitrol van database systemen en vergelijking van beschikbaarheid

Engelse titel: Automatisch uitrol van database systemen en vergelijking van beschikbaarheid

UDC: 681.3

Korte inhoud:

Hier komt een heel bondig abstract van hooguit 500 woorden. LaTeX commando's mogen hier gebruikt worden. Blanco lijnen (of het commando `\par`) zijn wel niet toegelaten!

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

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

Promotor: Prof. dr. ir. Wouter Joosen

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

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