HO
GENT

Faculteit Departement IT en Digitale Innovatie

High Availability oplossingen voor PostgreSQL: een vergelijkende studie en proof of concept

Elias Ameye

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Thomas Aelbrecht
Co-promotor:
Ruben Demey

Instelling: ST Engineering iDirect

Academiejaar: 2020-2021

Tweede examenperiode

Faculteit Departement IT en Digitale Innovatie

High Availability oplossingen voor PostgreSQL: een vergelijkende studie en proof of concept

Elias Ameye

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Thomas Aelbrecht
Co-promotor:
Ruben Demey

Instelling: ST Engineering iDirect

Academiejaar: 2020-2021

Tweede examenperiode

Woord vooraf

Deze bachelorproef werd geschreven in het kader van het voltooien van de opleiding Toegepaste Informatica afstudeerrichting Systeem- en Netwerkbeheer. Ik heb gekozen voor dit onderwerp omdat High Availability clustering mij tijdens de opleiding altijd al interesseerde. High Availability lijkt mij iets dat steeds meer en meer relevant wordt. Zeker met de huidige corona-crisis is een stabiele online omgeving cruciaal om competitief te blijven. Dit onderwerp gaf mij de kans om mijzelf 100% te verdiepen in een onderwerp dat mij interesseerde.

Ik wil graag Thomas Aelbrecht, mijn promotor, bedanken voor de goede begeleiding en de duidelijke feedback. Ongeveer tweewekelijks kwamen we samen om eens te overlopen hoever ik zat. Hierdoor gaf ik mijzelf telkens een deadline tegen wanneer ik bepaalde zaken verricht wou hebben.

Ook wil ik Ruben Demey, co-promotor, bedanken om bij de vragen die ik had, duidelijke antwoorden te geven waardoor ik telkens een stap dichter was bij het einde.

Ik wil ook Shavawn Somers bedanken om mijn bachelorproef helemaal door te nemen en een blik te werpen op de grammatica en leesbaarheid.

Zeker wil ik mijn ouders bedanken voor de financiële en mentale steun om deze opleiding tot een goed einde te brengen.

Tot slot wil ik ook nog mijn vriendin bedanken voor de vele steun en toeverlaat.

Veel leesplezier toegewenst!

Samenvatting

Dit onderzoek heeft tot doel na te gaan welke PostgreSQL High Availability cluster oplossingen momenteel voldoen aan de opgestelde requirements uit de bedrijfswereld. Het opzetten van een High Availability cluster wordt in deze tijden, waar het internet overheerst, steeds meer aangeraden. Ook PostgreSQL, als relationele databaseserver, neemt niet af in gebruik. In dit onderzoek komen verschillende PostgreSQL High Availability oplossingen aan bod en worden deze aan de hand van een requirementanalyse met elkaar vergeleken.

Gebruikmakende van deze requirements, zou het opzetten en beheren van een PostgreSQL High Availability cluster minder ingewikkeld moeten zijn. Deze oplossing zal garanderen dat een gebruiker of bedrijf zich minder zorgen zal moeten maken in geval van downtime van de database.

Aan de hand van die bevindingen wordt een ultieme oplossing gekozen waarmee een proof of concept wordt opgezet.

Het onderzoek zal een antwoord bieden op de vraag welke huidige PostgreSQL High Availability oplossingen geschikt zijn om replicatie, monitoring en failover te garanderen in een cluster. Deze vergelijkende studie kan een meerwaarde bieden aan bedrijven die werken met een kleine of grote PostgreSQL cluster waarin zij High Availability willen implementeren. Het kan ook een meerwaarde zijn voor bedrijven die al High Availability implementaties hebben in hun cluster, maar die een frisse blik nodig hebben, of willen upgraden naar een meer hedendaagse oplossing. Dit onderzoek richt zich bijgevolg op systeem- en netwerkbeheerders die dagelijks bezig zijn met het onderhouden en/of beheren van servers, meer specifiek database servers. Dit onderzoek kan ook een voorzet zijn voor toekomstige soortgelijke onderzoeken, waar gekeken kan worden of de huidige

oplossingen nog voldoen aan de verwachtingen over High Availability clustering in de toekomst.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1. Inleiding

1.1 Probleemstelling

In dit onderzoek zal de focus liggen op High Availability oplossingen en het belang ervan in PostgreSQL clusters. Met de Coronapandemie die nog overal aanwezig is, is er een grote stijging in het digitale gebruik bij klanten en bedrijven. Online winkelen heeft een enorme groei gekend. Koerierbedrijven hebben overuren moeten draaien om pakjes en post te brengen bij de mensen thuis. Technologie bedrijven kennen een extra druk omdat er meer beroep wordt gedaan op IT-services. Deze digitale (r)evolutie toont ons dat beschikbaarheid van diensten zeker heel relevant is. Stel je voor dat een server van Zalando, door software problemen, uitvalt. Alle aankopen van het laatste uur zijn niet doorgekomen. Een financieel drama. De oplossing? Een standby server die inspringt in geval van downtime. Resultaat? Geen downtime, geen financieel drama, geen geknoei met corrupte data. Met dit voorbeeld wordt op een eenvoudige manier het belang aangetoond van High Availability in database clusters. Stel, er loopt iets mis, kan het probleem snel opgelost worden, zonder dat de klant of het bedrijf er nadelige ervaringen aan overhoudt.

1.2 Onderzoeksvraag

Dit onderzoek focust op de vraag: Welke PostgreSQL High Availability cluster oplossing(en) kunnen bedrijven gebruiken om garantie te hebben op monitoring, replicatie en failover?

In dit onderzoek zullen vier verschillende PostgreSQL High Availability cluster oplossingen vergeleken worden met elkaar om een antwoord te kunnen geven op deze onderzoeks-

vraag. In Hoofdstuk 10 zal dit onderzoek een antwoord geven op deze onderzoeksvraag.

1.3 Onderzoeksdoelstelling

Het beoogde resultaat van deze bachelorproef is om uit de vergelijkende studie één oplossing voor te stellen die kan gebruikt worden voor de implementatie van een High Available PostgreSQL cluster. Hieraan gekoppeld wordt ook een eerste poging tot een proof of concept getoond waarin deze oplossing geïmplementeerd zit. Het belangrijkste in dit onderzoek zal zijn wanneer er een PostgreSQL High Availability cluster oplossing is die voldoet aan de gevraagde requirements. Wanneer deze oplossing gevonden is, zal dit onderzoek geslaagd zijn.

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie. Hierbij ligt de focus op High Availability, PostgreSQL, clustering en de reeds aanwezige oplossingen voor High Availability.

In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen. Hierin worden de functionele en niet-functionele requirements aan de hand van de MoSCoW-methode geprioriteerd om dan met elkaar vergeleken te worden. Hierbij zal elke oplossing aan de hand van deze requirements punten krijgen. De oplossing met de meeste punten zal gebruikt worden bij het opstellen van de proof of concept.

In Hoofdstuk 4, Hoofdstuk 5, Hoofdstuk 6 en Hoofdstuk 7 worden de vier oplossingen besproken en wordt gekeken of deze oplossingen aan de verschillende requirements voldoen.

In Hoofdstuk 8 zullen de resultaten van de oplossingen, afgetoetst aan de hand van de requirements, verder onderzocht worden. Hieruit wordt dan één oplossing gekozen die gebruikt zal worden voor de opzet van de proof of concept, te vinden in Hoofdstuk 9. Hierin zullen de verschillende stappen doorlopen worden en zijn code snippets terug te vinden die gebruikt zijn geweest bij het opbouwen van de cluster. Hierbij zal ook een persoonlijke conclusie te vinden zijn over eigen ervaringen bij het opzetten van deze PostgreSQL High Availability cluster.

Tot slot wordt in Hoofdstuk 10 de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2. Stand van zaken

Dit onderzoek is een vergelijkende studie tussen verschillende PostgreSQL oplossingen. In dit hoofdstuk zullen al verschillende oplossingen aan bod komen. De bedoeling van dit hoofdstuk is een inleiding in de verschillende termen en methodes die in dit onderzoek zullen gebruikt worden.

2.1 Databank

Oracle (2021) omschrijft een databank als een georganiseerde verzameling van gestructureerde informatie, of gegevens, die meestal elektronisch in een computersysteem wordt opgeslagen. Een database wordt gewoonlijk beheerd door een databasebeheersysteem (DBMS). Samen worden de gegevens en het DBMS, samen met de toepassingen die ermee verbonden zijn, een databasesysteem genoemd, vaak afgekort tot gewoon database (Oracle, 2021).

2.1.1 Relationele Databank

Een relationele databank is een type databank waarin gebruik wordt gemaakt van een structuur die het mogelijk maakt om gegevens te identificeren en te benaderen in relatie tot een ander deeltje data in diezelfde databank. Deze gegevens worden vaak georganiseerd in tabellen. Deze tabellen kunnen honderden, duizenden, miljoenen rijen en kolommen aan data hebben. Een kolom kent vaak ook een specifiek gegevenstype. Deze gegevenstypes kunnen getallen (integers), woorden (strings) of andere soorten bevatten (Codecademy, 2018).

2.1.2 Object-georiënteerde databank

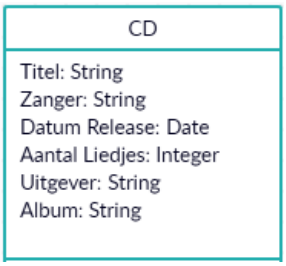
Een objectgeoriënteerde database (OODBMS) is een type databank die zich baseert op objectgeoriënteerd programmeren (OOP). De gegevens worden hier voorgesteld en opgeslagen in de vorm van objecten. OODBMS worden ook objectdatabases of objectgeoriënteerde databasemanagementsystemen genoemd (CSharp-Corner, 2019).

2.1.3 SQL

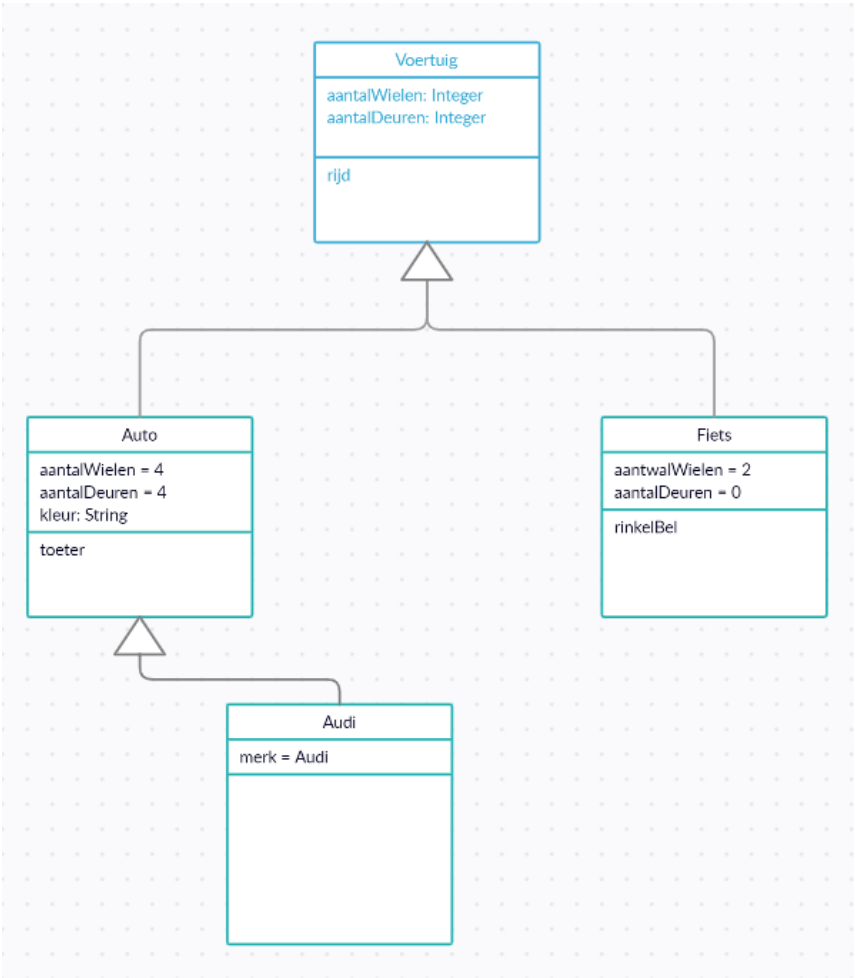
SQL (structured query language) is de eigen taal specifiek ontwikkeld voor interactie met databanken. Een databank modelleert entiteiten uit het echte leven en slaat deze op in tabellen. Via SQL is het mogelijk om de gegevens in deze tabellen te manipuleren (Carchedi, 2020). SQL wordt door alle relationele databanken gebruikt zoals MySQL, PostgreSQL, OracleDB, SQLite... (Codecademy, 2018).

2.1.4 Object-relationele databank

Een object-relationele database (ORD / ORDBMS) is een samenstelling uit zowel een relationele database (RDB / RDBMS), als een object-georiënteerde database (OOD / OODBMS). Samen ondersteunt het de basiscomponenten van elk objectgeoriënteerd databasemodel in zijn schema's en gebruikte querytaal, zoals klassen, overerving en objecten. Het bevat aspecten en kenmerken van bovenstaande genoemde modellen. Zo wordt het relationele duidelijk in de manier van opslaan van gegevens. Deze worden opgeslagen in een traditionele database en worden dan met behulp van SQL query's gemanipuleerd en benaderd. Aan de andere kant is ook het objectgeoriënteerde gedeelte merkbaar, namelijk dat de database beschouwd wordt als een objectopslag. Kort gezegd is één van de voornaamste doelstellingen van een object-relationele database, het dichten van de kloof tussen relationele en objectgeoriënteerde modelleringstechnieken en conceptuele datamodelleringstechnieken zoals daar zijn het entiteit-relatiediagram (ERD) en object-relationeel mappen (ORM) (Technopedia, 2021b). Klasses, overerving, types en functies zijn kenmerken van de object-relationele database en zullen de basisconcepten vormen voor PostgreSQL. Een klasse is een verzameling van gegevenstypes die bij eenzelfde soort iets horen. Bijvoorbeeld een klasse CD kan als kenmerken hebben: titel, zanger, datum uitgave, aantal liedjes... . Overerving is wanneer een klasse bepaalde kenmerken overerft of krijgt van een superklasse. Dit wordt geïllustreerd in onderstaande figuur 2.2. Hierbij wordt als voorbeeld een klasse Fiets gebruikt, die overerving doet klasse Vervoermiddel. Hierbij krijgt de klasse Fiets de nodige kenmerken, overgeërfd, van de klasse Vervoermiddel. Hierin is de klasse Fiets een specialisatie van de klasse Vervoermiddel. Zo is een kenmerk van de klasse Vervoermiddel het aantal wielen. Dit datatype wordt dan overgeërfd naar de klasse Fiets, waar de invulling van dit datatype twee is. Bij de klasse Auto heeft het overgeërfde kenmerk vier wielen. Een type is hierboven al vermeld. Dit gaat over de verschillende soorten data die er zijn. Getallen, woorden, objecten zijn hier voorbeelden van. Functies zijn een reeks SQL-statements die een specifieke taak uitvoeren. Functies bevorderen de herbruikbaarheid van code.



Figuur 2.1: Voorbeeld klasse CD



Figuur 2.2: Voorbeeld overerving

2.2 PostgreSQL

PostgreSQL is een open source systeem dat zich toelegt op het beheer van object-relationele databases. Het heeft meer dan 30 jaar actieve ontwikkeling en heeft een sterke reputatie op vlak van betrouwbaarheid, robuustheid van functies en prestaties (postgresql, 2021). PostgreSQL biedt een uitgebreide set van functionaliteiten die een hoge mate van customisatie mogelijk maakt binnen het systeem. Dit gaat van data administratie, beveiliging, tot back-up en herstel . PostgreSQL wordt regelmatig bijgewerkt door de PostgreSQL

Global Development Group en bijdragers uit de community. Deze community ondersteunt zichzelf en zijn gebruikers door het aanbieden van online educatieve bronnen en communicatiekanalen, zoals daar zijn PostgreSQL wiki, online forums en officiële documentatie. Er zijn ook bedrijven die commerciële support bieden aan een prijs (Nethosting, 2019).

Volgens DB-Engines is PostgreSQL de vierde database die vandaag de dag het meest gebruikt wordt en de tweede meest gebruikte open source database, na MySQL (DB-Engines, 2021a). DB-Engines verklaarde PostgreSQL in 2017, 2018 en 2020 het DBMS (Database management system) van het jaar (DB-Engines, 2021b). PostgreSQL biedt veel mogelijkheden om ontwikkelaars te helpen bij het bouwen van applicaties; om beheerders te helpen bij het beschermen van data-integriteit; en bij het bouwen van fouttolerante omgevingen. Het helpt ook bij het beheren van data, hoe groot of hoe klein de dataset ook is. PostgreSQL voldoet sinds september 2020 aan 170 van de 179 verplichte functies voor SQL:2016 Core conformiteit. Schaalbaarheid valt ook toe te schrijven aan PostgreSQL, dit zowel in de hoeveelheid data die het kan beheren, als in het aantal gelijktijdige gebruikers dat het kan accomoderen. Er zijn actieve PostgreSQL-clusters in productie omgevingen die terabytes aan data beheren, en gespecialiseerde systemen die zelfs petabytes beheren (PostgreSQL, 2021a). Deze laatste twee voorbeelden tonen hoe schaalbaar PostgreSQL wel is en kan zijn.

Postgres speelt in op de bovenvernoemde vier basisconcepten van een object-relationele databank zodat gebruikers het systeem makkelijk kunnen uitbreiden. Deze vier kenmerken, naast nog andere functies maken van Postgres een object-relationele database. Bovenstaande vermelde kenmerken zouden doen blijken dat Postgres voornamelijk een object-georiënteerde database is, maar de ondersteuning van de traditionele relationele databases, toont duidelijk aan dat, ondanks de object-georiënteerde kenmerken, Postgres stevig verankerd is in de relationele database wereld (PostgreSQL, 2021d).

2.3 High Availability

Het doel van High Availability architectuur is ervoor te zorgen dat een server, website of applicatie verschillende soorten netwerkbelastingen en verschillende soorten storingen kan verdragen. En dit met de minst mogelijke downtime. Door gebruik te maken van best practices die zijn ontworpen om hoge beschikbaarheid te garanderen, helpt dit volgens ServersAustralia (2017) om in een organisatie maximale productiviteit en betrouwbaarheid te bereiken (ServersAustralia, 2017).

High Availability is het vermogen van een systeem om continu operationeel te blijven gedurende een lange tijd. Men kan Availability meten ten opzichte van 100% operationeel, als in, nooit uitvallen. Beschikbaarheid wordt vaak uitgedrukt als een percentage van uptime in een bepaald jaar op basis van de SLA's, Service Level Agreements (Kumar, 2020). Vaak duidt men deze norm aan als de Five 9's, namelijk 99,999% beschikbaarheid (Lutkevich, 2021). High availability impliceert dat delen van een systeem volledig zijn getest en dat er voorzieningen zijn voor storingen/failures in de vorm van redundante

componenten. Servers kunnen worden ingesteld om in geval van nood de verantwoordelijkheden over te dragen aan een externe server, in een back-up proces. Hier spreekt men dan van failover (Technopedia, 2021a). Dit aspect zal verder belicht worden.

Wanneer is High Availability nu eigenlijk nodig? High Availability is nuttig wanneer bedrijven te maken hebben met dagelijk kritisch applicatiebeheer. Ook wanneer er veel verkeer is op de website, is het niet te veroorloven om downtime te hebben. Daarnaast is gewoon een goede service aanbieden, ook een geldige reden om aan High Availability te doen (CriticalCase, 2020).

Belangrijke principes van High Availability zijn:

1. **Het elimineren van single point of failure**: Toevoeging van redundantie zorgt ervoor dat het falen van een onderdeel in het systeem niet leidt tot het volledige falen van een geheel systeem.

2. **Betrouwbare cross-over**: In een redundant systeem wordt het kruispunt zelf een single point of failure. Fouttolerante systemen moeten voorzien in een betrouwbaar crossover- of automatisch omschakelingsmechanisme om storingen te voorkomen.

3. **Storingsdetectie**: Als bovenstaande principes proactief bewaakt worden, dan zal een gebruiker misschien nooit een systeemstoring zien. Postgres biedt de bouwstenen om bovenstaande principes volledig uit te werken zodat er op deze manier High Availability verzekerd kan worden.

In geval van een volledige uitval van een systeem is geografische redundantie algemeen zeer wenselijk. Op deze manier worden servers verdeeld over meerdere locaties over de wereld. Bij downtime door een natuurramp bijvoorbeeld zijn standby servers op meerdere fysieke (ongetroffen) locaties beschikbaar om in te vallen. Dit type van redundantie kan zeer duur uitdraaien, waarbij het een verstandige beslissing kan zijn om te kiezen voor een gehoste oplossing, waarbij de provider datacenters heeft over heel de wereld.

2.3.1 Load Balancing

Load Balancing is ook een manier om High Availability te waarborgen. Het doel van een load balancer is om toepassingen en/of netwerkverkeer te verdelen over meerdere servers en componenten. Het zal binnenkomende verzoeken routeren naar verschillende servers. Hiermeer wil het prestaties en betrouwbaarheid verbeteren. Een voorbeeld van load balancing is Round Robin die ervoor zorgt dat de verzoeken van de load balancer naar de eerste server gaan in de rij. De verzoeken gaan deze rij af, tot hij op het einde komt, waarna hij terug van het eerste element in de rij begint. Een andere manier van load balancing is Least Connection. Hierbij zal er gekozen worden om gebruik te maken van de server met het minst aantal actieve verbindingen. Load balancers spelen een rol bij het tot stand brengen van een infrastructuur met High Availability, maar het hebben van een load balancer staat niet garant voor het hebben van High Availability. Door redundantie te implementeren voor de load balancer zelf, kan deze geëlimineerd worden als een single

point of failure (Jevtic, 2018))

HAProxy, High Availability Proxy, is hier een voorbeeld van. Dit is een op software gebaseerde TCP/HTTP load balancer. Het verdeelt de werklast over meerdere servers zodat het prestaties kan maximaliseren en het gebruik van resources kan optimaliseren (SeveralNines, 2020).

2.3.2 Schaalbaarheid

Schaalbaarheid is de eigenschap van een systeem om te kunnen voldoen aan een groeiend aantal eisen door bronnen toe te voegen. De redenen voor een groei kunnen tijdelijk zijn, bijvoorbeeld wanneer een bedrijf net een nieuw product op de markt brengt, of wanneer er een substantiële groei is in het aantal klanten of personeel. Er zijn twee manieren om aan schalen te doen: horizontaal en verticaal (Insausti, 2019).

Horizontaal schalen

Horizontaal schalen richt zich op het toevoegen van nodes in een systeem of cluster. Hierin is het handig om meer slave nodes toe te voegen aan de cluster. Dit zal helpen om lees- en schrijfprestaties te verbeteren, doordat het meer gebalanceerd kan gebeuren. Hierbij is een load balancer nodig, die dit verkeer zal regelen. Eén load balancer zal niet voldoende zijn om single point of failure preventief te voorkomen. Beter hier is dan twee of meerdere load balancers toevoegen. Op deze manier blijft High Availability een garantie (Insausti, 2019).

Verticaal schalen

Verticaal schalen is het toevoegen van hardware resources aan bestaande nodes. Deze resources kunnen gaan over RAM-geheugen, schijfgeheugen, CPU (Insausti, 2019).

2.3.3 Failover

Failover is het (automatisch) overschakelen naar een back-upsysteem. Wanneer een primair systeemonderdeel faalt, wordt failover ingeschakeld om de negatieve gevolgen te elimineren of te beperken (AVINetworks, 2020).

2.3.4 Failback

Failback is het proces van het herstellen van operaties naar een primaire machine nadat ze zijn verschoven geweest naar een secundaire machine wegens failover (TechTarget, 2020).

2.4 Cluster oplossingen

2.4.1 Cluster

Een cluster is een groepering van servers die met elkaar samenwerken om één geheel te vormen. Op deze manier kan een cluster High Availability mogelijk maken (TechTarget, 2017).

2.4.2 Standby mogelijkheden

Bij het elimineren van single points of failure ondersteunt Postgres de volgende fysieke stand-by's:

1. Cold Standby: Dit is een back-up server die beschikt over back-ups en alle nodige WAL-bestanden voor herstel. WAL is de afkorting voor Write Ahead Log. Het logt elke transactie die uitgevoerd wordt op een database voordat het wordt uitgevoerd (PostgreSQL, 2021e). Een Cold Standby systeem is geen operationeel systeem, maar het kan wel beschikbaar worden gemaakt wanneer dit nodig geacht wordt. Voornamelijk worden dan back-up servers en WAL bestanden gebruikt voor het maken van een nieuwe PostgreSQL node als onderdeel van disaster recovery.

2. Warm Standby: Hierin draait Postgres in herstelmodus en ontvangt het updates door gebruik te maken van gearchiveerde logbestanden of door gebruik te maken van log shipping replicatie van Postgres. Log shipping is een proces waarbij de back-up van transactielogbestanden op een primaire database wordt geautomatiseerd en vervolgens op een standby server wordt hersteld (Roth, 2016). In deze modus aanvaardt Postgres geen verbindingen en queries.

3. Hot Standby: Ook bij Hot Standby draait Postgres in herstelmodus en ontvangt het updates door gebruik te maken van gearchiveerde logbestanden of door gebruik te maken van log shipping van Postgres. Het verschil met Warm Standby is dat in deze herstelmodus Postgres wel verbindingen ondersteunt en read-only queries.

Bovenstaande voorbeelden zijn mogelijkheden die kunnen helpen bij het elimineren van single points of failure. Afhankelijk van het overeengekomen niveau van beschikbaarheid, kunnen gebruikers voor een van de bovenstaande kiezen.

2.4.3 Patroni

Patroni is een open source cluster-technologie, geschreven in Python die zich focust op automatische failover en High Availability voor een PostgreSQL databank. Het dient als een soort cluster manager die de implementatie en het onderhoud van High Availability in PostgreSQl-clusters zal aanpassen en automatiseren. Het maakt gebruik van gedistribueerde configuratieopslagplaatsen zoals etcd, Consul, ZooKeeper of Kubernetes voor maximale toegankelijkheid (Markwort, 2018).

Patroni biedt cloud-native netwerkfuncties en geavanceerde opties voor failback en failover. Een cloud-native netwerkfunctie is een software-implementatie van een netwerkfunctie, die wordt uitgevoerd in een linux-container, die traditioneel wordt uitgevoerd door een fysiek apparaat (CDNF, 2020).

Als oplossing zorgt Patroni voor een end-to-end setup van High Available PostgreSQL-clusters, inclusief streaming replicatie. Streaming replicatie biedt de mogelijkheid om continu WAL logs naar standby servers te sturen en deze toe te passen om de servers op deze manier up to date te houden (wiki.postgresql.org, 2020). Het biedt verschillende manieren waarop een standby node kan aangemaakt worden, en dient als sjabloon dat kan worden aangepast naar wat gewenst wordt.

Bij het aanmaken van een standby node maakt Patroni gebruik van pg_basebackup en ondersteunt het ook methodes zoals Barman en pgBackRest. die gebruikt worden voor het aanmaken van standby nodes.

Na het opzetten van de cluster, zal Patroni zich actief toeleggen op monitoren. Patroni zal werken aan de hand van een leader lock. Deze wordt om de zoveel tijd vernieuwd, en wanneer de master node er niet in slaagt om deze te vernieuwen, zal Patroni een nieuwe node verkiezen tot master node (ScaleGrid, 2018a).

2.4.4 Pgpool-II

Pgpool-II omschrijft zichzelf als een middleware die werkt tussen PostgreSQL servers en een PostgreSQL database client. Het ondersteunt High Availability, geautomatiseerde load balancing, etc. Pgpool-II biedt ook logical replication. Logische replicatie, of logical replication stelt gebruikers in staat om een selectieve replica van bepaalde tabellen uit te voeren en deze uit te schrijven naar een standby node. Met logical replication kan een standby node replicatie ingeschakeld hebben van meerdere master nodes. Dit kan handig zijn in situaties waar je gegevens van verschillende PostgreSQL databases moet repliceren naar een enkele PostgreSQL server voor rapportage en data warehousing. Eén van de grootste voordelen van logische replication boven streaming replication is dat logical toelaat om veranderingen van een oudere versie van PostgreSQL naar een nieuwere versie te repliceren. Streaming replicatie werkt alleen wanneer zowel de master- als de standby node van dezelfde versie zijn. Wat niet altijd ideaal is in grote opstellingen (Ihalainen, 2019).

Pgpool-II biedt de volgende features:

1. Connection Pooling

Pgpool-II bewaart verbindingen naar de PostgreSQL servers, en hergebruikt ze wanneer een nieuwe verbinding met dezelfde eigenschappen zoals gebruikersnaam, database of protocol versie binnenkomt. Dit vermindert de overhead van verbindingen, en verbetert de totale doorvoer van het systeem.

2. Replication

Pgpool-II kan meerdere PostgreSQL servers beheren. Door gebruik te maken van de replicatiefunctie kan een realtime back-up worden gemaakt op twee of meerdere fysieke schijven, zodat de dienst kan worden voortgezet zonder servers te stoppen in geval van een schijfstoring.

3. Load Balancing

Als een database wordt gerepliceerd, zal het uitvoeren van een query op elke server hetzelfde resultaat opleveren. Pgpool-II maakt gebruik van de replicatie mogelijkheid om de belasting op elke PostgreSQL server te verminderen door queries over meerdere servers te verdelen, waardoor de totale throughput van het systeem verbetert. In het meest gunstige geval verbetert de prestatie evenredig met het aantal PostgreSQL servers. Load balance werkt het beste in een situatie waarin veel gebruikers zijn die veel queries tegelijkertijd uitvoeren.

4. Limiting Exceeding Connections

Er is een limiet op het maximum aantal gelijktijdige verbindingen met PostgreSQL. Verbindingen die gemaakt willen worden nadat dit maximum aantal verbindingen overschreden is, worden geweigerd. Het instellen van een maximum aantal verbindingen verhoogt echter het verbruik van bronnen en beïnvloedt de systeemprestaties. Pgpool-II heeft ook een limiet op het maximum aantal verbindingen, maar extra verbindingen worden hier niet geweigerd, maar worden in een wachtrij geplaatst, zodat wanneer een verbinding verdwijnt, de volgende zijn plaats kan innemen.

5. Watchdog

Watchdog kan een robuust clustersysteem creëren en het single point of failure of split brain vermijden. Watchdog kan worden ingesteld op de nodes en kan zo een lifecheck uitvoeren op de Pgpool-II nodes, om dan op deze manier fouten in Pgpool-II te detecteren. Wanneer een actieve Pgpool-II node down gaat, kan dan een standby Pgpool-II node gepromoveerd worden tot actief, en zal deze dan virtueel het IP overnemen.

6. In Memory Query Cache

In memory query cache maakt het mogelijk om een paar SELECT statements en zijn resultaat op te slaan. Wanneer een identieke SELECT binnenkomt, retourneert Pgpool-II de waarde uit de cache. Omdat er geen SQL parsing of toegang tot PostgreSQL aan te pas komt, is het gebruik van in memory cache extreem snel. Aan de andere kant kan het in sommige gevallen langzamer zijn dan het normale pad, omdat het wat overhead toevoegt van het opslaan van cache gegevens.

Pgpool-II praat met de backend en de frontend protocollen van PostgreSQL, en legt een verbinding tussen beide. Daarom denkt een database applicatie (frontend) dat Pgpool-II de eigenlijke PostgreSQL server is, en de server (backend) ziet Pgpool-II als een van zijn clients. Omdat Pgpool-II transparant is voor zowel de server als de client, kan een bestaande databasetoepassing met Pgpool-II worden gebruikt vrijwel zonder de broncode aan te passen (pgpool, 2021).

2.4.5 PostgreSQL Automatic Failover (PAF)

PostgreSQL Automatic Failover (PAF) is een nieuwe resource agent, speciaal voor PostgreSQL. Door Pacemaker en Corosync, is het voor PAF mogelijk om:

- aan detectiestoring te doen van een PostgreSQL instance.
- de primary server te herstellen
- aan failover te doen naar een standby server.
- om de best (met de kleinste vertraging) beschikbaarste standby server te selecteren bij failover.
- rollen te wisselen in de cluster tussen standby en primary nodes.

De oorspronkelijke wens is om een duidelijke grens te houden tussen de Pacemaker administratie en de PostgreSQL administratie, om de cluster eenvoudig, gedocumenteerd en toch krachtig te houden.

Zodra een PostgreSQL-cluster is opgebouwd met behulp van interne streaming replicatie, is PAF in staat om aan Pacemaker te laten zien wat de huidige status is van de PostgreSQL instantie op elke node: primary, standby, gestopt, etc. Mocht dan een storing optreden op de primary node, dan zal Pacemaker standaard proberen deze te herstellen. Mocht de storing niet te herstellen zijn, dan zorgt PAF ervoor dat de standby servers de beste van hen kunnen kiezen (de dichtstbijzijnde bij de oude primary) en deze promoveren tot de nieuwe primary. Dit doet PAF dankzij Pacemaker (de Rorthais, 2020).

Postgres Automatic Failover (PAF) biedt verschillende voordelen in het omgaan met PostgreSQL High Availability. PAF gebruikt IP adres failover in plaats van het herstarten van de standby om verbinding te maken met de nieuwe master tijdens een failover event. Dit blijkt voordelig te zijn in scenario's waarbij de gebruiker de standby nodes niet automatisch terug wilt toevoegen aan de cluster. PAF heeft ook zeer weinig manuele tussenkomst nodig en beheert de algemene gezondheid van alle Postgres databankbronnen. Het enige geval waarin manuele tussenkomst een vereiste is, is dan in het geval van een tijdlijn data divergentie waarbij de gebruiker kan kiezen om pg_rewind te gebruiken (ScaleGrid, 2018b). Pg_rewind is een tool die helpt bij het synchroniseren van een PostgreSQL-cluster met een andere kopie van diezelfde cluster, met als enige verschil de tijd. Voorbeeld hierbij is een oude master node terug online brengen na failover als een standby node die de nieuwe master node volgt (PostgreSQL, 2021c).

2.4.6 Replication Manager (repmgr)

Repmgr is een open source oplossing die zich focust op het beheren van replicatie en failover van servers in PostgreSQl-clusters. Het verbetert de ingebouwde hot-standby opties van PostgreSQL met extra features zoals tools om standby servers op te zetten, replicatie te monitoren en administratieve taken uit te voeren, zoals failover (repmgr, 2021b).

De features die repmgr 5 aanbiedt zijn:

1. De implementatie als een PostgreSQL extentie.

2. Replicatie cluster monitoring.

3. Standby klonen aan de hand van pg_basebackup of Barman

- pg_basebackup: pg_basebackup wordt gebruikt om basisback-ups te maken van een draaiende PostgreSQL databank cluster. Deze back-ups worden gemaakt zonder de aanwezige clients, die in verbinding staan met de databank, te beïnvloeden. Deze back-ups kunnen gebruikt worden voor zowel point-in-time recovery, maar ook als een startpunt voor log shipping of streaming replicatie standby servers. Bij point-in-time recovery wordt verwezen naar herstel van dataveranderingen tot een bepaald punt in de tijd (MySQL, 2021).
  pg_basebackup maakt een binaire kopie van de database cluster bestanden, terwijl het ervoor zorgt dat het systeem automatisch in en uit back-up modus wordt gezet. Back-ups worden altijd gemaakt van de gehele databasecluster; het is niet mogelijk om een back-up te maken van afzonderlijke databases of databaseobjecten.
  De back-up wordt gemaakt over een gewone PostgreSQL verbinding maakt gebruik van het replicatieprotocol. De server moet ook worden geconfigureerd om ten minste één sessie beschikbaar te laten voor de back-up (PostgreSQL, 2021b).

- Barman: Barman of pgbarman staat voor Backup en Recovery Manager. Het is een open source beheertool voor disaster recovery van PostgreSQL servers. Het is geschreven in Python. Barman laat toe om op afstand van meerdere servers in bedrijfskritische omgevingen back-ups uit te voeren. Het helpt ook database beheerders tijdens een herstelfase (Barman, 2020b) (Barman, 2020a).

4. Promotie standby server

Standby server die kan worden gepromoveerd tot een primary server zonder herstart. Andere standby servers die verbinding kunnen maken met de nieuwe master zonder opnieuw gesynchroniseerd te worden (wiki.postgresql, 2020).

5.Cascading Standby Support

Standby servers die niet direct verbonden zijn met de master node worden niet beïnvloed tijdens failover van de primary naar een andere standby mode.

6. Vereenvoudigen van het beheer van WAL-retentie en ondersteuning voor replicatiesleuven

Door de vereenvoudiging van WAL-retentie zal het dus eenvoudiger zijn om WAL-bestanden op te ruimen vanaf elke bestandssysteemlocatie. Ook in standby servers kan het gebruikt worden om bestanden die niet meer nodig zijn, te verwijderen uit de standby server (Augustine, 2019). Een sleuf vertegenwoordigt een stroom van wijzigingen ten opzichte van één database (Ranganathan, 2020).

7. Switchover ondersteuning voor rolswitching tussen primary en standby

Hierin wordt een standby server gepromoveerd tot een primary server en zal deze primary server degraderen naar een standby server. Wanneer andere standby servers verbonden zijn met de degradatiekandidaat, kan repmgr deze instrueren om de nieuwe primary server te volgen en niet de oude, die net gedegradeerd is (repmgr, 2021a).

2.4.7 PgBouncer

PgBouncer is een open source connection-pooler die gebruikt wordt voor PostgreSQL. Het kan verbindingen naar één of meerdere databases poolen. Het onderhoudt voor een pool van verbindingen per gebruiker. Het wordt vaak geconfigureerd om één van de verbindingen uit te delen aan een nieuwe inkomende cliëntverbinding en terug te geven aan de pool wanneer de cliënt de verbinding verbreekt (Ramachandran, 2019).

2.4.8 Raima

Databasereplicatie is het proces waarin gegevens worden gekopieerd van een database naar één of meerdere replica's. Dit om de toegankelijkheid van gegevens en fouttolerantie te verbeteren. In de context van replicatie gebruikt men vaak ook de termen actief-actief en actief-passief. Raima Database Manager (RDM) ondersteunt beide technieken. Bij Raima wordt actieve replicatie gewoon replicatie genoemd en passieve replicatie spiegelen (mirroring). Spiegelen zal resulteren in identieke replica's zoals de originele database, terwijl replicatie zal resulteren in replica's die niet identiek zijn aan de originele database. Deze replica's zullen alle records bevatten die van de originele database zijn overgebracht, maar de fysieke organisatie van de records in de databasebestanden (of in het geheugen) kan verschillen. Om terug te komen op de termen actief-actief en actief-passief zullen die vaker verwijzen naar andere concepten dan dewelke juist omschreven (replicatie en spiegelen). Actieve-actieve replicatie betekent replicatie in twee richtingen van gegevens tussen twee databases die beide actief worden bijgewerkt. Actieve-passieve replicatie betekent replicatie in één richting van een actief bijgewerkte master node naar een slave node die niet wordt bijgewerkt, behalve door het replicatieproces. Hier verwijst men soms ook naar master-slave replicatie. In RDM is replicatie altijd actief-passief (Raima, 2021).

2.4.9 EDB

Voor betrouwbare cross-over biedt EntrepriseDB (EDB) een technologie genaamd EDB Postgres Failover Manager (EFM). Dit maakt automatische failover van de Postgres master node naar een standby node mogelijk in geval van een software- of hardwarefout op de master. EFM maakt gebruik van JGroups, die een betrouwbare, gedistribueerde en redundante infrastructuur biedt zonder een single point of failure. EDB Postgres Failover Manager kan ook gebruikt worden voor de detectie van storingen. Het bewaakt de server continu en zal storingen op verschillende niveaus detecteren. Het is ook capabel om om failover uit te voeren van de master node naar één van de replica nodes om het systeem beschikbaar te maken voor het accepteren van databaseverbindingen en queries. Wanneer EFM goed geconfigureerd is, kan het storingen detecteren en direct failover uitvoeren.

Verlies van service kan in twee categorieën opdelen. Geplande uitval of downtime en ongeplande uitval of downtime. Geplande downtime is vaak het gevolg van onderhoudsactiviteiten. Dit kan zijn door een softwarepatches die een herstart van het systeem of van de database vereist. In het algemeen is deze uitval niet onverwachts en zal deze uitval geen grootschalige gevolgen hebben. Een ongeplande downtime is vaak het resultaat van een of andere fysieke gebeurtenis, zoals hardware- of softwarestoring, of een anomalie in de omgeving. Stroomuitval, defecte CPU- of RAM-componenten (of eventueel andere hardwarecomponenten), netwerkstoringen, inbreuken op de beveiliging, of diverse defecten in toepassingen, middleware en besturingssystemen resulteren bijvoorbeeld in ongeplande uitval. In geval van (on)geplande downtime kan EFM helpen om de downtime zoveel mogelijk te minimaliseren. Voor een geplande downtime kan een gebruiker bijvoorbeeld eerst alle standby nodes patchen en EFM gebruiken om over te schakelen voordat de master node gepatcht wordt. Bij een ongeplande downtime kan EFM ervoor zorgen dat de storingen gedetecteerd worden en failover uitvoeren naar de juiste standby node, om dan deze node de nieuwe master node te maken. EFM zal na dit proces er ook voor zorgen dat de oude master node niet terugkomt om een split-brain situatie te voorkomen. Split-brain duidt op de inconsistenties in beschikbaarheid en data. Hierdoor ontstaan twee afzonderlijke datasets met overlap (Kumar, 2020).

3. Methodologie

Dit onderzoek start met een diepgaande en extensieve literatuurstudie over PostgreSQL, High Availability en de reeds bestaande High Available PostgreSQL cluster oplossingen. Deze literatuurstudie is terug te vinden in Hoofdstuk 2.

Na de literatuurstudie zal worden geduid hoe de verschillende High Available PostgreSQL cluster oplossingen geschift zullen worden. In deze schifting zullen verschillende requirements opgezet worden waaraan de verschillende oplossingen zullen afgetoetst worden. Dit zal gebeuren aan de hand van een requirementsanalyse waarin de verschillende requirements aan bod zullen komen. Deze requirements zullen dan aan de hand van de MoSCoW-methode geprioriteerd worden om zo het belang van elke requirement in te schatten. Deze analyse zal van groot belang zijn in dit onderzoek aangezien ze de basis zal vormen waarop er een High Available PostgreSQL cluster oplossing zal beoordeeld en gekozen worden.

Na de verwerking van de resultaten zal er in dit onderzoek één oplossing gebruikt worden om een proof of concept uit te werken. Hier zal dan gekeken worden om de verschillende functionele requirements te implementeren in de cluster. Op deze manier zal dan worden aangetoond dat deze oplossing een volwaardig PostgreSQL High Availability cluster oplossing is voor bedrijven.

3.1 Requirementsanalyse

Elk van deze High Available PostgreSQL cluster oplossingen zal worden afgetoetst aan de requirements om ze op deze manier te evalueren. In samenspraak met Ruben Demey,

Global IT Operations Manager bij ST Engineering iDirect, zijn er drie functionele requirements bepaald. Bij elke oplossing worden de requirements afgetoetst en krijgen ze een score van 1 (slecht) tot 5 (uitstekend).

3.1.1 Functionele Requirements

Een functionele requirement beschrijft hoe het systeem moet werken en wat het moet kunnen. Het gaat om de volgende elementen:

- Ondersteuning van replicatie
- Ondersteuning van failover
- Ondersteuning van monitoring

3.1.2 Niet-functionele Requirements

Een niet-functionele requirement is een kwaliteitseis voor het systeem. Hier worden vooral voorkeuren mee bedoeld. Hier gaat het om volgende elementen:

- Open source
- Actieve ondersteuning in 2020-2021
  Een belangrijke requirement is dat de oplossing up-to-date moet zijn. In dit onderzoek zal er geen gebruik gemaakt worden van achterhaalde, oude tools. Het is belangrijk dat de oplossing futureproof is.
- Grafische interface
  Een gemakkelijk te gebruiken interface is zeker een meerwaarde bij het monitoren en beheren van een cluster. Als er geen specialist aanwezig is, kan de grafische interface soms voor wat extra duidelijkheid zorgen.
- Beperkte manuele interventie
  Een cluster opzetten die nadien niet veel manuele interventie nodig heeft, is ideaal in situaties waar er weinig tot geen kennis is over de cluster. Tijdens en na downtime zal de server automatisch acties uitvoeren die het verdere bestaan van de cluster garanderen.

3.2 Indelen requirements volgens MoSCoW-techniek

Na het opstellen van de verschillende requirements kunnen deze nog eens ingedeeld worden volgens de MoSCoW-techniek (Ahmad e.a., 2017). Hierin worden de requirements geprioriteerd in 4 categorieën. Must have, Should have, Could have, Won't have. Must have-requirements zijn verplicht aanwezig. Should have-requirements zijn geen verplichting, maar zijn het liefst wel aanwezig in de keuze van een oplossing. Could have-requirements zijn volledig optioneel en zijn dus niet geheel relevant bij het kiezen van de oplossing. Het Won't have aspect wordt hier achterwege gelaten, omdat dit geen invloed zal hebben op de keuze van een oplossing. Wont have-requirements zijn diegene met de

laagste prioriteit. De onderverdeling verloopt als volgt:

- **Must have**
  - Ondersteuning van replicatie
  - Ondersteuning van failover
  - Ondersteuning van monitoring
- **Should have**
  - Actieve ondersteuning in 2020-2021
  - Open source
- **Could have**
  - Grafische interface
  - Beperkte manuele interventie

4. Patroni

In dit hoofdstuk wordt dieper ingegaan op Patroni als een PostgreSQL High Availability cluster oplossing. De verschillende requirements, waaraan voldaan moet worden, worden geanalyseerd. Hierbij wordt dan telkens ook een score gegeven die in Hoofdstuk 8 verder belicht zullen worden.

4.1 Inleiding tot Patroni

Patroni, als oplossing, is ontstaan uit een fork van Governor, een project van Compose. Het is een in Python geschreven open source oplossing ontwikkeld door Zalando. Het beheert High Availability van PostgreSQL clusters.

4.2 Requirements

4.2.1 Must have

Ondersteuning van replicatie

Voor het kopiëren van bestanden of databases, of replicatie genaamd maakt Patroni gebruikt van streaming replicatie. Standaard maakt Patroni gebruik van asynchrone replicatie. Hierbij worden gegevens eerst geschreven naar een primaire opslagarray. Vervolgens worden de te repliceren gegevens gecommitteerd naar het geheugen. Daarna worden deze gegevens in real-time of met geplande tussenpozen naar de replicatiedoelen gekopieerd. Wanneer gewerkt wordt met asynchrone replicatie, kan de cluster het zich permitteren om

een aantal gecommitteerde transacties te verliezen, om High Availability te garanderen. Wanneer de primary node faalt, en Patroni een standby node regelt, zullen alle transacties die niet naar de standby node zijn gerepliceerd, verloren gaan. Het aantal transacties dat verloren mag gaan, kan geregeld worden aan de hand van maximum_lag_on_failover. Dit zal bij failover het aantal verloren transacties beperken.

PostgreSQL synchrone replicatie is ook mogelijk bij Patroni. Synchrone replicatie zorgt voor meer consistentie van data in een cluster door te bevestigen dat schrijfacties naar een secundary node worden geschreven voordat ze naar de verbindende client terugkeren met een succes. De nadelen van synchrone replicatie houden in dat er een verminderde verwerkingscapaciteit is voor schrijfacties. Deze verwerkingscapaciteit is volledig gebaseerd op netwerkprestaties. Het gebruik van PostgreSQL synchrone replicatie garandeert niet dat er onder alle omstandigheden geen transacties verloren zullen gaan. Wanneer de primary node en de secundary node die op dat moment synchrone replicatie draaien, gelijktijdig falen, zal een derde node, die mogelijk niet alle transacties bevat, worden gepromoveerd tot primary node.

Voor replicatie opties bestaan er verschillende tools zoals barman en Wal-E, die zullen helpen om nodes toe te voegen aan de cluster. Er kan ook gekozen worden van waar sommige nodes hun data zullen halen.

Hieruit blijk dat Patroni zeer flexibel en eenvoudig te gebruiken is bij het configureren van replicatie in een High Availability cluster. Op basis hiervan krijgt deze requirement een score van 4.

Ondersteuning van failover

Patroni clusters kennen een automatische failover wanneer de primary node (onverwachts) niet beschikbaar is. De failover opties kunnen zeer geavanceerd aangepast worden naar de wensen van de cluster. Wanneer, na failover, een primary node terug online komt, kan deze aan de hand van pg_rewind terug toegevoegd worden aan de cluster als standby node. Patroni werkt aan de hand van endpoints. Dit kunnen switchover of failover endpoints zijn. Een failover endpoint laat toe om een manuele failover uit te voeren wanneer er in de cluster geen 'gezonde' nodes meer aanwezig zijn. Deze endpoint wordt gebruikt door patronictl failover. Aan de hand van dynamische configuratie instellingen, opgeslagen in het Distributed Configuration Store (DCS), die geldig zijn voor alle nodes, worden parameters zoals ttl, maximum_lag_on_failover en retry_timeout. ingesteld. Bij ttl wordt dan de tijd ingesteld die nodig is om de leader lock in te nemen als standby node. Normaal bevat de primary node altijd deze lock, maar als er iets misloopt en de primary node door één of andere reden deze lock misloopt, kan een standby node deze leader lock overpakken en op deze manier zichzelf promoten tot primary node en zo failover initiëren. Patroni kan bijvoorbeeld worden verteld om nooit een standby node te promoten die meer dan een configureerbare hoeveelheid log achterloopt op de primary node.

Op deze requirement doet Patroni het zeer goed. Door de hierboven genoemde redenen krijgt Patroni voor failover een 4.

Ondersteuning van monitoring

Patroni kent een zeer uitgebreide REST API die gebruikt kan worden om de cluster en zijn nodes te monitoren en voor applicaties om te selecteren met welke PostgreSQL instantie verbinding moet worden gemaakt. Aan de hand van Consul kan je monitoring aanzetten in de cluster. Dit moet dan wel op elke node aangezet worden. Er zijn ook tools beschikbaar waarbij monitoring de enige functionaliteit is. PGWatch/PGWatch2 zijn hier voorbeelden van. PGWatch maakt gebruik van data die al verzameld wordt in PostgreSQL queries. Dus toevoegen of wijzigen van deze queries kan zeer eenvoudig gebeuren.

Ook op deze requirement scoort Patroni redelijk goed. Er is een uitgebreid aanbod beschikbaar om monitoring te voorzien in de cluster. Hiervoor krijgt Patroni opnieuw een 4.

4.2.2 Should have

Actieve ondersteuning in 2020-2021

Er worden nog steeds releases uitgerold. Op het moment van schrijven dateerde de laatste release van 22 februari 2021. Deze release bevatte nieuwe features waaronder toegevoegde support voor de REST API bij TLS keys en cipher suite limitations. Deze release bevatte ook nog stabiliteitsverbetering en bugfixes.

Op basis hiervan krijgt Patroni voor deze requirement een score van 5.

Open source

Patroni is volledig open source. De source code is online te vinden op github.com en wordt nog tot op vandaag geüpdatet.

Patroni is een open source oplossing, dus voldoet zeker ook aan deze requirement. Hierbij krijgt het ook een score van 5.

4.2.3 Could have

Grafische interface

Patroni kent ook grafische interfaces die gebruikt kunnen worden bij het opzetten van een Patroni PostgreSQL High Availability cluster. Een voorbeeld hiervan is Patroni Environment Setup (PES). Hiermee kan er op Windows eenvoudig, snel en gebruikersvriendelijk een Patroni cluster opgezet worden.

Beperkte manuele interventie

Patroni kan opgesteld worden op een manier waarbij niet veel manuele interventie nodig is.

Dit was de laatste requirement waarop Patroni geëvalueerd zou worden. In onderstaande tabel 4.1 is het overzicht terug te vinden van de scores per requirement. Dit zal enkel gaan over de requirements must have en should have. De could have requirements hebben geen invloed op de eindscore.

Overzicht score Patroni	
Must have	
Ondersteuning van replicatie	4
Ondersteuning van failover	4
Ondersteuning van monitoring	4
Should have	
Open Source	5
Ondersteuning in 2020-2021	5
Totaal	22/25

Tabel 4.1: Overzicht score Patroni

5. PostgreSQL Automatic Failover (PAF)

In dit hoofdstuk wordt meer informatie verleend over PostgreSQL Automatic Failover (PAF) als een PostgreSQL High Availability cluster oplossing. Er zal worden ingegaan op de verschillende requirements, waaraan voldaan moet worden. Hierbij wordt ook een score gegeven die in Hoofdstuk 8 verder geanalyseerd zal worden.

5.1 Inleiding tot PostgreSQL Automatic Failover (PAF)

PostgreSQL Automatic Failover (PAF) is een High Availability oplossing ontwikkeld door ClusterLabs die zich bezighoudt met het beheer van High Availability in PostgreSQL clusters. Het maakt vooral gebruik van Postgres synchrone replicatie om te garanderen dat er geen gegevens verloren gaan op het moment van de failover operatie.

PostgreSQL Automatic Failover (PAF) werkt nauw samen met de tool Pacemaker. PAF is in staat om aan Pacemaker te laten zien wat de huidige status is van een node. Bij het optreden van een storing zal Pacemaker automatisch proberen dit te herstellen. Als de storing niet te herstellen valt, dan zal PAF zorgen voor automatische failover

Pacemaker is een service die in staat is om veel resources te beheren, en doet dit met de hulp van hun resource agents. Resource agents hebben de verantwoordelijkheid om een specifieke resource af te handelen en Pacemaker te informeren over deze resultaten.

5.2 Requirements

5.2.1 Must have

Ondersteuning van replicatie

PostgreSQL Automatic Failover (PAF) zal niet 100% kunnen beschermen tegen gegevensverlies. De replicatie wordt geconfigureerd met PostgreSQL. Bij asynchrone replicatie, wat de standaard is voor PostgreSQL, zullen de transacties eerst op primary node gecommitteerd worden voordat ze op de standby nodes zullen worden toegepast. In het geval van failover, zal de meest up-to-date standby node gepromoot worden door PostgreSQL Automatic Failover (PAF). Hierdoor zal gegevensverlies geminimaliseerd worden, maar niet onbestaande zijn.

Voornamelijk zal PostgreSQL Automatic Failover (PAF) gebruik maken van synchrone replicatie om te garanderen dat er geen data verloren gaat tijdens een failover.

Hierdoor krijgt PostgreSQL Automatic Failover (PAF) voor deze requirement een score van 4.

Ondersteuning van failover

Aan de hand van Pacemaker kan er een automatische failover plaatsvinden. PostgreSQL Automatic Failover (PAF) communiceert continu met Pacemaker over de status van de cluster en bewaakt het functioneren van de database. In geval van storing, wordt Pacemaker hierover geïnformeerd en zal het kijken om de storing op te lossen. Is de storing onoplosbaar, dan zal Pacemaker een verkiezing starten tussen de standby nodes om een opvolger van de primary node te selecteren. Bij het configureren van de cluster wordt elke node geconfigureerd als standby node vooraleer Pacemaker bepaalt welke node de primary wordt. Dit zorgt er ook voor dat elke node weet hoe hij moet functioneren als een standby node.

master-max krijgt bij de configuratie van Pacemaker het aantal PostgreSQL nodes dat als primary zullen dienen op een gegeven moment. Default staat deze ingesteld op 1.

Bij clone-max staat de ingestelde parameter gelijk aan het aantal nodes dat PostgreSQL kunnen draaien, primary of standby. Deze parameter staat normaal gelijk aan het aantal nodes die aanwezig zijn in de cluster.

notify=true laat toe om te signaleren wanneer er bepaalde acties gebeuren op de node. Dit zal in contact staan met Pacemaker. Deze staat standaard op inactief, dus moet de parameter liefst expliciet nog eens vermeld worden op 'true'.

PAF maakt gebruik van ip-adres failover in plaats van het herstarten van de standby node om verbinding te maken met de nieuwe master tijdens een failover event, wat voordelig is in scenario's waar een gebruiker de standby nodes niet wil herstarten.

Door de hierboven genoemde redenen krijgt PostgreSQL Automatic Failover (PAF) een score van 4 voor deze requirement.

Ondersteuning van monitoring

Via de tool Pacemaker kan er bij PostgreSQL Automatic Failover (PAF) goed aan monitoring gedaan worden. Het zal signaleren wanneer een node niet bereikbaar is. Het toevoegen van Watchdog als tool is hier mogelijk om ook monitoring te doen.

Hierdoor krijgt PostgreSQL Automatic Failover (PAF) voor monitoring een score van 4.

5.2.2 Should have

Actieve ondersteuning in 2020-2021

De laatste release dateert van 10 maart 2020. Hieruit kan worden geconcludeerd dat ook deze oplossing nog up-to-date is en bruikbaar voor dit onderzoek.

Hier krijgt PostgreSQL Automatic Failover (PAF) een score van 4. Een score van 5 ging behaald worden moest de laatste release voor PostgreSQL Automatic Failover (PAF) in 2021 zijn.

Open source

PostgreSQL Automatic Failover (PAF) is een open source High Availability oplossing voor PostgreSQL clusters.

Voor deze requirement krijgt PostgreSQL Automatic Failover (PAF) een score van 5.

5.2.3 Could have

Grafische interface

Op eerste zicht lijkt er geen algemeen gekende grafische interface aanwezig te zijn voor PostgreSQL Automatic Failover (PAF).

Beperkte manuele interventie

PostgreSQL Automatic Failover (PAF) kan zo geconfigureerd worden dat het weinig tot geen manuele interventie vereist. Aan de hand van bijvoorbeeld Pacemaker is dit mogelijk.

Dit was de laatste requirement waarop PostgreSQL Automatic Failover (PAF) geëvalueerd zou worden. In onderstaande tabel 5.1 is het overzicht terug te vinden van de scores

per requirement. Dit zal enkel gaan over de requirements must have en should have. De could have requirements hebben geen invloed op de eindscore.

Overzicht score PostgreSQL Automatic Failover (PAF)	
Must have	
Ondersteuning van replicatie	4
Ondersteuning van failover	4
Ondersteuning van monitoring	4
Should have	
Open Source	5
Ondersteuning in 2020-2021	4
Totaal	21/25

Tabel 5.1: Overzicht score PostgreSQL Automatic Failover (PAF)

6. Pgpool-II

In dit hoofdstuk wordt meer informatie aangeleverd over Pgpool-II als een PostgreSQL High Availability cluster oplossing. Er zal worden ingegaan op de verschillende requirements, waaraan voldaan moet worden. Hierbij wordt dan telkens ook een score gegeven die in Hoofdstuk 8 verder geanalyseerd zal worden.

6.1 Inleiding tot Pgpool-II

Pgpool-II is een soort van proxy software die zit tussen PostgreSQL nodes en PostgreSQL databases. Het biedt onder andere ondersteuning voor High Availability en automatische load balancing.

6.2 Requirements

6.2.1 Must have

Ondersteuning van replicatie

Pgpool-II beschikt over een tool, genaamd Watchdog, die een subproces is van Pgpool-II dat dient om High Availability toe te voegen. Watchdog wordt gebruikt om single points of failure op te lossen door meerdere Pgpool-II nodes te coördineren. Het coördineert meerdere Pgpool-II nodes door informatie met elkaar uit te wisselen. Op die manier kan het zeker zijn dat de database service onaangetast blijft.

Standaard voor PostgreSQL zal er gebruik worden gemaakt van streaming replication bij Pgpool-II.

Voor deze redenen krijgt Pgpool-II een score van 3 voor replicatie.

Ondersteuning van failover

Pgpool-II voorziet ook automatische failover binnenin Watchdog. Pgpool-II kan ook andere soorten van failover configuraties aan zoals failover_requires_consensus of failover_when_quorum_exists.

Bij Pgpool-II, wanneer het failover_command geconfigureerd is, zal deze automatisch tijdens een failover worden uitgevoerd. Hierin zal een nieuwe primary node uit de bestaande standby nodes gekozen worden. Ook kan dit commando signalen uitsturen, zoals een mail om te laten weten dat er failover plaatsvindt binnen de cluster.

Pgpool-II krijgt voor deze requirement een score van 4.

Ondersteuning van monitoring

Watchdog kan ook remote de gezondheid controleren van de node waarop het is geïnstalleerd door de verbinding met upstream nodes te monitoren. Als de monitoring faalt, behandelt watchdog dit als het falen van een lokale Pgpool-II node en zal het de nodige acties ondernemen.

Pgpool-II kan meerdere nodes beheren en hierin replicatie opzetten om High Availability te verkrijgen

Voor monitoring krijgt Pgpool-II een score van 3.

6.2.2 Should have

Actieve ondersteuning in 2020-2021

De laatste release van Pgpool-II vond plaats op 26 november 2020. Hieruit kan worden besloten dat Pgpool-II nog een geldige, bruikbare oplossing is.

Voor deze requirement krijgt Pgpool-II een score van 4. Een score van 5 zou zijn wanneer Pgpool-II in 2021 een release zou gehad hebben.

Open source

Pgpool-II is een open source project. De source code wordt onderhouden door hun git-repository.

Pgpool-II krijgt voor deze requirement een score van 5.

6.2.3 Could have

Grafische interface

Op eerste zicht is geen algemeen gekende grafische interface aanwezig voor Pgpool-II.

Beperkte manuele interventie

Via de tool Watchdog is het mogelijk om bepaalde elementen in Pgpool-II te automatiseren. Automatische failover is hier een voorbeeld van. Andere elementen zijn echter wel nog manueel te configureren.

Dit was de laatste requirement waarop Pgpool-II geëvalueerd zou worden. In onderstaande tabel 6.1 is het overzicht terug te vinden van de scores per requirement. Dit zal enkel gaan over de requirements must have en should have. De could have requirements hebben geen invloed op de eindscore.

Overzicht score Pgpool-II	
Must have	
Ondersteuning van replicatie	3
Ondersteuning van failover	4
Ondersteuning van monitoring	3
Should have	
Open Source	5
Ondersteuning in 2020-2021	4
Totaal	19/25

Tabel 6.1: Overzicht score Pgpool-II

7. Replication Manager (repmgr)

In dit hoofdstuk wordt meer informatie verleend over Replication Manager (repmgr) als een PostgreSQL High Availability cluster oplossing. Er zal worden ingegaan op de verschillende requirements, waaraan voldaan moet worden. Hierbij wordt dan telkens ook een score gegeven die in Hoofdstuk 8 verder geanalyseerd zal worden.

7.1 Inleiding tot Replication Manager (repmgr)

Replication Manager (repmgr) is een open source oplossing ontwikkeld door 2ndQuadrant. Het is ontwikkeld voor het beheer van replicatie en failover in een PostgreSQL cluster.

Replication Manager (repmgr) biedt de tools om PostgreSQL replicatie op te zetten, te configureren, te beheren en te monitoren. Het stelt de gebruiker in staat om handmatige omschakeling en failover taken uit te voeren met behulp van repmgr utility. Dit is een open source tool die PostgreSQL's ingebouwde streaming replicatie ondersteunt en verbetert.

7.2 Requirements

7.2.1 Must have

Ondersteuning van replicatie

Replication Manager (RepMgr) maakt standaard gebruik van streaming replicatie in zijn clusters.

Deze streaming replicatie moet vaak manueel geconfigureerd worden in zowel de primary als de standby node. Het zal ook gebruik maken van pg_basebackup zodat alle standby nodes kunnen invallen wanneer de primary node uitvalt.

Met repmgr is het mogelijk om deze taken geautomatiseerd uit te voeren, wat veel tijd en resources kan sparen.

Voor replicatie krijgt Replication Manager (repmgr) een score van 4.

Ondersteuning van failover

Aan de hand van de tool repmgrd zal er automatische failover in gang worden gezet wanneer de primary node op down komt te staan. Hier wordt er dan een nieuwe standby node gekozen invalt voor de voormalige primary node.

Het zal in geval van failover of andere situaties ook de nodige notificaties uitsturen om te laten weten wat er aan de hand is. Deze situaties worden dan door de gebruiker zelf geregistreerd.

Om repmgrd te laten werken zal in het repmgr.conf bestand, bij failover 'automatic' worden ingegeven. Hierna zal de repmgrd daemon de cluster actief beginnen te monitoren. Als er een fout optreedt in de primary node, zal het meerdere keren proberen opnieuw verbinding te maken. Als alle pogingen om verbinding te maken met de primary node mislukken, wordt de meest geschikte standby gekozen als de nieuwe primary node door repmgrd.

Door de hierboven genoemde redenen, krijgt Replication Manager (repmgr) een score van 4 voor failover.

Ondersteuning van monitoring

Over Monitoring is er bij Replication Manager (repmgr) niet veel te vinden. De tools repmgrd zal wel actief de cluster monitoren, en op basis hiervan de nodige acties uitvoeren, naargelang de staat van de cluster.

Voor monitoring krijgt Replication Manager (repmgr) een score van 3.

7.2.2 Should have

Actieve ondersteuning in 2020-2021

De laatste release voor Replication Manager (repmgr) was op 22 oktober 2020. Replication Manager (RepMgr) krijgt hierdoor een score van 4/5 voor Should have. Doordat er (nog) geen release is in 2021, krijgt het geen 5/5.

Open source

Replication Manager (repmgr) is een open source High Availability tool voor PostgreSQL clustering.

7.2.3 Could have

Grafische interface

Op eerste zicht lijkt er niet direct een bekende grafische interface aanwezig te zijn voor Replication Manager (repmgr).

Beperkte manuele interventie

Bij Replication Manager (repmgr) is er nog veel dat manueel moet uitegevoerd worden.

Dit was de laatste requirement waarop Replication Manager (repmgr) geëvalueerd zou worden. In onderstaande tabel 7.1 is het overzicht terug te vinden van de scores per requirement. Dit zal enkel gaan over de requirements must have en should have. De could have requirements hebben geen invloed op de eindscore.

Overzicht score Replication Manager (repmgr)	
Must have	
Ondersteuning van replicatie	4
Ondersteuning van failover	4
Ondersteuning van monitoring	3
Should have	
Open Source	5
Ondersteuning in 2020-2021	4
Totaal	20/25

Tabel 7.1: Overzicht score Replication Manager (repmgr)

8. Verwerking resultaten

In dit hoofdstuk zullen de verschillende resultaten van de requirementanalyse van de oplossingen verwerkt worden om zo tot een statistisch correct antwoord te komen op de hoofdonderzoeksvraag uit Sectie 1.2.

8.1 Resultaten requirements

Hieronder vindt u per PostgreSQL High Availability cluster oplossing een overzicht van het gemiddelde, de mediaan, de modus, de standaardafwijking en de variatiecoëfficiënt van alle scores gegeven op de requirements.

	Patroni	PostgreSQL Automatic Failover (PAF)	Pgpool-II	Replication Manager (repmgr)
Gemiddelde	4.4	4,2	3,8	4
Mediaan	4	4	4	4
Modus	4	4	3 & 4	4
Standaardafwijking	0,55	0,45	0,84	0,71
Variatiecoëfficiënt	0,12	0,11	0,22	0,18

Tabel 8.1: Statistieken requirements

Om deze resultaten met elkaar te vergelijken, wordt er gebruik gemaakt van niet-gepaarde t-testen. Hierin stellen we μ_1 gelijk aan het populatiegemiddelde van de scores van Patroni en μ_2 aan het populatiegemiddelde van PostgreSQL Automatic Failover (PAF). Op deze manier willen we aantonen dat Patroni beter scoort dan PostgreSQL Automatic Failover (PAF). We nemen als $\alpha = 5\%$. In Figuur 8.1 wordt deze hypothese berekend in R.

Uit deze t-test komt een p-waarde van 0.2727 wat hoger ligt dan het significatieniveau

van 0,05. Hierdoor moeten we onze nulhypothese aanvaarden en stellen dat Patroni niet significant beter scoort dan PostgreSQL Automatic Failover (PAF).

```
> # Scoort Patroni beter dan PostgreSQL Automatic Failover (PAF)?
> t.test(scorePatroni, scorePAF, paired = FALSE, alternative= "greater", mu=0)

        Welch Two Sample t-test

data:  scorePatroni and scorePAF
t = 0.63246, df = 7.6923, p-value = 0.2727
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 -0.3911046        Inf
sample estimates:
mean of x mean of y
      4.4       4.2

>
```

Figuur 8.1: R berekening Patroni en PostgreSQL Automatic Failover (PAF)

Wanneer we kijken naar Patroni en Pgpool-II, dan kunnen we opnieuw een niet-gepaarde t-test opstellen. Hierin stellen we μ_1 gelijk aan het populatiegemiddelde van de scores van Patroni en μ_2 aan het populatiegemiddelde van Pgpool-II. Op deze manier willen we aantonen dat Patroni beter scoort dan Pgpool-II. We nemen als $\alpha = 5\%$. In Figuur 8.2 wordt deze hypothese berekend in R.

Uit deze t-test komt een p-waarde van 0.1111 wat hoger ligt dan het significatieniveau van 0,05. Hierdoor moeten we onze nulhypothese aanvaarden en stellen dat Patroni niet significant beter scoort dan Pgpool-II.

```
> # Scoort Patroni beter dan Pgpool-II?
> t.test(scorePatroni, scorePool, paired = FALSE, alternative= "greater", mu=0)

        Welch Two Sample t-test

data:  scorePatroni and scorePool
t = 1.3416, df = 6.8966, p-value = 0.1111
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 -0.2491973        Inf
sample estimates:
mean of x mean of y
      4.4       3.8

>
```

Figuur 8.2: R berekening Patroni en Pgpool-II

Wanneer we kijken naar Patroni en Replication Manager (repmgr), dan kunnen we opnieuw een niet-gepaarde t-test opstellen. Hierin stellen we μ_1 gelijk aan het populatiegemiddelde van de scores van Patroni en μ_2 aan het populatiegemiddelde van Replication Manager (repmgr). Op deze manier willen we aantonen dat Patroni beter scoort dan Replication Manager (repmgr). We nemen als $\alpha = 5\%$. In Figuur 8.3 wordt deze hypothese berekend in R.

Uit deze t-test komt een p-waarde van 0.1742 wat hoger ligt dan het significatieniveau van 0,05. Hierdoor moeten we onze nulhypothese aanvaarden en stellen dat Patroni niet significant beter scoort dan Replication Manager (repmgr).

```
> # Scoort Patroni beter dan Replication Manager(repmgr)?
> t.test(scorePatroni, scoreRep, paired = FALSE, alternative= "greater", mu=0)

        Welch Two Sample t-test

data:  scorePatroni and scoreRep
t = 1, df = 7.5294, p-value = 0.1742
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 -0.349891       Inf
sample estimates:
mean of x mean of y
      4.4       4.0

> |
```

Figuur 8.3: R berekening Patroni en Replication Manager (repmgr)

Wanneer we verder kijken naar PostgreSQL Automatic Failover (PAF) en Pgpool-II, dan kunnen we opnieuw een niet-gepaarde t-test opstellen. Hierin stellen we μ_1 gelijk aan het populatiegemiddelde van de scores van PostgreSQL Automatic Failover (PAF) en μ_2 aan het populatiegemiddelde van Pgpool-II. Op deze manier willen we aantonen dat PostgreSQL Automatic Failover (PAF) beter scoort dan Pgpool-II. We nemen als $\alpha = 5\%$. In Figuur 8.4 wordt deze hypothese berekend in R.

Uit deze t-test komt een p-waarde van 0.1908 wat hoger ligt dan het significatieniveau van 0,05. Hierdoor moeten we onze nulhypothese aanvaarden en stellen dat PostgreSQL Automatic Failover (PAF) opnieuw niet significant beter scoort dan Pgpool-II.

```
> # Scoort PostgreSQL Automatic Failover (PAF) beter dan Pgpool-II?
> t.test(scorePAF, scorePool, paired = FALSE, alternative= "greater", mu=0)

        Welch Two Sample t-test

data:  scorePAF and scorePool
t = 0.94281, df = 6.1132, p-value = 0.1908
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 -0.4216954        Inf
sample estimates:
mean of x mean of y
      4.2       3.8

> |
```

Figuur 8.4: R berekening PostgreSQL Automatic Failover (PAF) en Pgpool-II

Kijken we naar PostgreSQL Automatic Failover (PAF) en Replication Manager (repmgr), dan kunnen we opnieuw een niet-gepaarde t-test opstellen. Hierin stellen we μ_1 gelijk aan het populatiegemiddelde van de scores van PostgreSQL Automatic Failover (PAF) en

μ_2 aan het populatiegemiddelde van Replication Manager (repmgr). Op deze manier willen we aantonen dat PostgreSQL Automatic Failover (PAF) beter scoort dan Replication Manager (repmgr). We nemen als $\alpha = 5\%$. In Figuur 8.5 wordt deze hypothese berekend in R.

Uit deze t-test komt een p-waarde van 0.3051 wat hoger ligt dan het significatieniveau van 0,05. Hierdoor moeten we onze nulhypothese aanvaarden en stellen dat PostgreSQL Automatic Failover (PAF) opnieuw niet significant beter scoort dan Replication Manager (repmgr).

```
> # Scoort PostgreSQL Automatic Failover (PAF) beter dan Replication Manager (repmgr)?
> t.test(scorePAF, scoreRep, paired = FALSE, alternative= "greater", mu=0)

        Welch Two Sample t-test

data:  scorePAF and scoreRep
t = 0.53452, df = 6.7586, p-value = 0.3051
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 -0.512714       Inf
sample estimates:
mean of x mean of y
      4.2       4.0

> |
```

Figuur 8.5: R berekening PostgreSQL Automatic Failover (PAF) en Replication Manager (repmgr)

Wanneer we verder kijken naar Replication Manager (repmgr) en Pgpool-II, dan kunnen we opnieuw een niet-gepaarde t-test opstellen. Hierin stellen we μ_1 gelijk aan het populatiegemiddelde van de scores van Replication Manager (repmgr) en μ_2 aan het populatiegemiddelde van Pgpool-II. Op deze manier willen we aantonen dat Replication Manager (repmgr) beter scoort dan Pgpool-II. We nemen als $\alpha = 5\%$. In Figuur 8.6 wordt deze hypothese berekend in R.

Uit deze t-test komt een p-waarde van 0.653 wat hoger ligt dan het significatieniveau van 0,05. Hierdoor moeten we onze nulhypothese aanvaarden en stellen dat Replication Manager (repmgr) opnieuw niet significant beter scoort dan Pgpool-II.

Bovenstaande resultaten werden behaald met onderstaande code in R die te zijn op Figuur 8.7, Figuur 8.8, Figuur 8.9 en Figuur 8.10. Hierin wordt telkens het gemiddelde, de mediaan, de modus, de standaardafwijking en de variatiecoëfficiënt berekend per oplossing.

```
> # Scoort Pgpool-II beter dan Replication Manager (repmgr)?
> t.test(scorePool, scoreRep, paired = FALSE, alternative= "greater", mu=0)

        Welch Two Sample t-test

data:  scorePool and scoreRep
t = -0.40825, df = 7.7838, p-value = 0.653
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 -1.11428      Inf
sample estimates:
mean of x mean of y
      3.8       4.0

>
```

Figuur 8.6: R berekening Replication Manager (repmgr) en Pgpool-II

```
> scorePatroni <- c(4,4,4,5,5)
>
> #Berekening gemiddelde, mediaan, standaardafwijking en variatiecoëfficiënt
>
> mean(scorePatroni)
[1] 4.4
> median(scorePatroni)
[1] 4
> sd(scorePatroni)
[1] 0.5477226
> sd(scorePatroni)/mean(scorePatroni)
[1] 0.1244824
>
```

Figuur 8.7: R berekening voor Patroni

```
> scorePAF <- c(4,4,4,5,4)
>
> #Berekening gemiddelde, mediaan, standaardafwijking en variatiecoëfficiënt
>
> mean(scorePAF)
[1] 4.2
> median(scorePAF)
[1] 4
> sd(scorePAF)
[1] 0.4472136
> sd(scorePAF)/mean(scorePAF)
[1] 0.1064794
>
```

Figuur 8.8: R berekening voor PostgreSQL Automatic Failover (PAF)

```
> scorePool <- c(3,4,3,5,4)
>
> #Berekening gemiddelde, mediaan, standaardafwijking en variatiecoëfficiënt
>
> mean(scorePool)
[1] 3.8
> median(scorePool)
[1] 4
> sd(scorePool)
[1] 0.83666
> sd(scorePool)/mean(scorePool)
[1] 0.2201737
>
```

Figuur 8.9: R berekening voor Pgpool-II

```
> scoreRep <- c(4,4,3,5,4)
>
> #Berekening gemiddelde, mediaan, standaardafwijking en variatiecoëfficiënt
>
> mean(scoreRep)
[1] 4
> median(scoreRep)
[1] 4
> sd(scoreRep)
[1] 0.7071068
> sd(scoreRep)/mean(scoreRep)
[1] 0.1767767
>
```

Figuur 8.10: R berekening voor Replication Manager (repmgr)

9. Proof of Concept

In dit hoofdstuk wordt de opbouw van de Patroni cluster uitgelegd. Hierin zal worden gekeken hoe de cluster voldoet aan de functionele requirements, namelijk replicatie, failover en monitoring. Als proof of concept is gekozen voor Patroni omdat uit de requirementanalyse deze oplossing het sterkst voldeed aan de requirements.

9.1 Omgeving

De opzet van deze proof of concept zal gebeuren in een lokale Linux-omgeving, namelijk Ubuntu 21.04. De cluster zal bestaan uit drie virtuele machines die zullen draaien op Ubuntu Xenial 16.04. De eerste virtuele machine (pgServer) is een servernode waarop Consul zal draaien, samen met pgBouncer en een HAProxy. De andere twee virtuele machines (pgNode1 en pgNode2) zullen PostgreSQL, Patroni en een Consul Agent draaien.

De cluster zal dus zo geconfigueerd worden dat er één primary node is met een standby node die asynchrone streaming replicatie verricht.

Ip-adressen	
pgServer	172.16.0.11
pgNode1	172.16.0.22
pgNode2	172.16.0.33

Tabel 9.1: Ip-adressen Cluster

9.2 Prerequisites

9.2.1 Vagrant

Bij de opzet van cluster zal gebruik worden gemaakt van Vagrant. Vagrant is een tool voor het bouwen en beheren van virtuele machine-omgevingen (Kalow, 2020).

In onderstaande code snippet is de code voor de Vagrantfile zichtbaar. Hierin wordt bepaald welke servers aangemaakt worden en welke opties zij bezitten. Zo zal elke node een RAM-geheugen hebben van 1 GB (1024 MB). Ook het ip-adres zal hierin worden toegewezen per node. Er wordt ook een bestand meegegeven dat zal worden uitgevoerd in de server. Dit bestand bevat de opbouw van de cluster, zoals het installeren van Consul, het configureren van Patroni op de nodes, en de setup van HAProxy en pgBouncer.

Bij de drie virtuele machines wordt telkens verwezen naar een Linux bash-script, namelijk postgresql-cluster-setup.sh. Dit is het bestand waarin wordt gekeken met welke virtuele machine er gewerkt wordt, zodat, op basis daarvan, de juiste packages geïnstalleerd kunnen worden. Zo zal bijvoorbeeld bij pgNode1 Python, Consul Agent, Patroni en PostgreSQL geïnstalleerd worden als packages.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
    config.vm.box = "ubuntu/xenial64"
    config.vm.provider "virtualbox" do |v|
        v.customize ["modifyvm", :id, "--memory", 1024]
        v.gui = true
    end

    config.vm.define :pgServer do |pgServer_config|
        pgServer_config.vm.hostname = 'pgServer'
        pgServer_config.vm.network :private_network, ip: "172.16.0.11"
        pgServer_config.vm.provision :shell, :path =>
            "postgresql-cluster-setup.sh"
    end
    config.vm.define :pgNode1, primary: true do |pgNode1_config|
        pgNode1_config.vm.hostname = 'pgNode1'
        pgNode1_config.vm.network :private_network, ip: "172.16.0.22"
        pgNode1_config.vm.provision :shell, :path =>
            "postgresql-cluster-setup.sh"
    end
    config.vm.define :pgNode2 do |pgNode2_config|
        pgNode2_config.vm.hostname = 'pgNode2'
        pgNode2_config.vm.network :private_network, ip: "172.16.0.33"
        pgNode2_config.vm.provision :shell, :path =>
```

```
            "postgresql-cluster-setup.sh"
    end
end
```

Bij het gebruik van Vagrant, komen er ook een paar vaak voorkomende commando's bij kijken. Zo is 'vagrant up' het commando dat gebruikt wordt om alle virtuele machines aan te zetten. Wanneer er één server alleen moet worden aangezet, kan er ook specifiek een virtuele machine aangeroepen worden: 'vagrant up pgnode1'. Dit zal enkel de virtuele machine met als naam pgnode1 opstarten. Een volgend veelgebruikt commando is 'vagrant provision'. Dit commando zal worden gebruikt wanneer er in de configuratiebestanden wijzigingen zijn toegebracht. Dit laat toe om deze wijzigingen uit te voeren op alle virtuele machines. Ook hier kan er specifiek één virtuele machine aangeroepen worden, dit is net zoals hierboven, door een specifieke naam mee te geven als parameter. 'vagrant halt' zal alle virtuele machines uitzetten. 'vagrant ssh pgnode1' zal ssh-connectie maken met de meegegeven virtuele machine als parameter.

Na het opzetten van onze Vagrantfile wordt deze via 'vagrant up' geactiveerd in een command line. Dit kan in Windows Powershell, maar ook in een Linux terminal of een andere soort terminal.

Vagrant is niet standaard geïnstalleerd op een Ubuntu machine. Via 'sudo apt-get install vagrant' wordt Vagrant geïnstalleerd.

9.2.2 VirtualBox

In deze proof of concept zal gebruik gemaakt worden van virtuele machines via VirtualBox. VirtualBox is een open source virtualisatiesoftware. Het werkt als een hypervisor en kan op deze manier virtuele machines aanmaken, waarin de gebruiker naar eigen keuze verschillende besturingssytemen kan emuleren. Bij de configuratie van een virtuele machine kunnen de specificaties zoals RAM-geheugen en schijfruimte bepaald worden.

9.3 pgServer

Tijdens de configuratie van de cluster zal pgServer '172.16.0.11' als ip adres hebben.

Als eerste stap moeten de juiste packages geïnstalleerd worden op pgServer. Hier gaat het over Python, Consul, pgBouncer en HAProxy. Dit zal gebeuren aan de hand van het postgresql-cluster-setup.sh bestand die wordt aangeroepen in de Vagrantfile.

Hierna zal voor Consul een service toegevoegd worden die bij het opstarten van de server wordt uitgevoerd. Belangrijk is dat bij het aanmaken van deze service, de service gestart en ge-enabled moet zijn. In deze service wordt verwezen naar het config bestand voor de Consul server. Deze ziet er als volgt uit.

```
{
    "advertise_addr": "172.16.0.11",
    "bind_addr": "172.16.0.11",
    "bootstrap": false,
    "bootstrap_expect": 1,
    "server": true,
    "client_addr": "127.0.0.1",
    "node_name": "pgServer",
    "datacenter": "dc1",
    "data_dir": "/var/consul/server",
    "domain": "consul",
    "encrypt": "/q/vkVS+My2nl8Zk/8csuQ==",
    "log_level": "INFO",
    "enable_syslog": true,
    "rejoin_after_leave": true,
    "ui_dir": "/var/consul/ui",
    "leave_on_terminate": false,
    "skip_leave_on_interrupt": false
}
```

Het advertise_addr of het adverteer adres wordt gebruikt om het IP-adres te wijzigen dat geadverteerd wordt naar de andere nodes. Standaard wordt voor advertising het bind_addr geadverteerd.

De bootstrap parameter toont aan of de server in bootstrap modus staat. bootstrap_expect geeft het aantal verwachte servers mee in de cluster als parameter. Consul zal wachten tot dit aantal beschikbaar is om de cluster te bootstrappen.

Het client_addr geeft het adres mee waaraan Consul client interfaces zal binden, zoals de HTTP en DNS servers. Standaard staat dit op 127.0.0.1.

datacenter controleert het datacenter waarin de Consul agent draait. Standaard staat deze ingesteld op dc1.

Consul zal aan de hand van rejoin_after_leave er altijd voor proberen zorgen dat de node bij opstart de cluster zal proberen te joinen. Standaard beschouwd Consul een vertrek van een node uit de cluster, als permanent. Met de parameter hier op true te zetten, zal dit niet het geval zijn en zal de node proberen de cluster terug te joinen.

leave_on_terminate zal, indien true, wanneer de node een signaal krijgt om af te sluiten, de node de cluster verlaten. Voor Consul clients staat dit standaard op true, maar bij Consul servers staat dit standaard op false. skip_leave_on_interrupt is zeer vergelijkbaar met leave_on_terminate.

Bij het installeren van de HAProxy wordt doorverwezen naar een configuratiebestand waarin de verschillende opties worden meegegeven.

```
global
```

```
    maxconn 100

defaults
    log global
    mode tcp
    retries 2
    timeout client 30m
    timeout connect 4s
    timeout server 30m
    timeout check 5s

listen stats
    mode http
    bind *:7000
    stats enable
    stats uri /

listen postgres
    bind *:5000
    option httpchk
    http-check expect status 200
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server postgresql_pgNode1_172.0.16.22 172.16.0.22:5432 maxconn 100
        check port 8008
    server postgresql_pgNode2_172.0.16.33 172.16.0.33:5432 maxconn 100
        check port 8008
```

maxconn zal eeen limiet zetten op het aantal verbindingen dat HAProxy zal aanvaarden.

mode zal bepalen op welke manier HAProxy werkt. Aan de ene kant kan het werken als een eenvoudige TCP proxy. Aan de andere kant is het ook in staat om HTTP-berichten van binnenkomend verkeer te inspecteren.

De timeout commando's zullen na een bepaalde time-out van client of server de verbinding verbreken.

bind zal een listener toewijzen aan een bepaald IP-adres en poort.

option httpchk zal gezondheidscontroles uitvoeren op HTTP. Servers die hier niet op reageren, zullen niet meer bediend worden.

De default-server configureert voor alle serverregels de standaardinstellingen, zoals het activeren van de gezondheidscontroles en maximale verbindingen.

Bij server staat voor elke aanwezig node het IP-adres, de poort en het maximum aantal connecties.

Bij het configureren van pgBouncer wordt gebruik gemaakt van een .ini file. Deze ziet er

als volgt uit:

```
[databases]
postgres = host=172.16.0.11 port=5000 pool_size=6
template1 = host=172.16.0.11 port=5000 pool_size=6
test = host=172.16.0.11 port=5000 pool_size=6

[pgbouncer]
logfile = /var/log/postgresql/pgbouncer.log
pidfile = /var/run/postgresql/pgbouncer.pid
listen_addr = *
listen_port = 6432
unix_socket_dir = /var/run/postgresql
auth_type = trust
auth_file = /etc/pgbouncer/userlist.txt
admin_users = postgres
stats_users =
pool_mode = transaction
server_reset_query =
server_check_query = select 1
server_check_delay = 10
max_client_conn = 1000
default_pool_size = 12
reserve_pool_size = 5
log_connections = 1
log_disconnections = 1
log_pooler_errors = 1
```

listen_addr specificeert het aantal IP-adressen waar naar geluisterd moet worden. Een * zal duiden dat naar alle adressen geluisterd zal worden. listen_port zal de poort meegeven waarnaar geluisterd moet worden.

auth_type zal bepalen hoe gebruikers geauthenticeerd worden. Bij de parameter trust zal er geen authenticatie plaatsvinden.

Bij admin_users zal worden bepaald welke gebruikers volledige controle hebben.

server_check_query controleert of de server connection aanwezig is. server_check_delay zal meegeven hoelang een verbinding beschikbaar moet blijven.

default_pool_size en reserve_pool_size zal het (maximum) aantal pools bepalen.

9.4 pgNode1

Tijdens de configuratie van de cluster zal pgNode1 ’172.16.0.22’ als ip adres hebben.

Idem als bij pgServer is de eerste stap het installeren van de juiste packages. Hier gaat

het over Python, Consul Agent, Patroni en PostgreSQL uiteraard. Dit zal opnieuw gebeuren aan de hand van het postgresql-cluster-setup.sh bestand die wordt aangeroepen in de Vagrantfile. Deze file is te vinden onderaan dit hoofdstuk in

Bij het configureren van Patroni zal verwezen worden naar pgNode1Patroni.yml waarin de configuratie te vinden is. Zoals eerder al vermeld is geweest, is de configuratie van Patroni te vinden in het DCS, het Distributed Configuration Store. Hierin staat onder andere de configuratie voor pg_rewind en maximum_lag_on_failover. Hieronder de configuratie:

```
scope: PatroniCluster
namespace: /nsPatroniCluster
name: pgNode1

log:
    level: INFO
    dir: /var/log/patroni
    file_size: 10485760 #staat gelijk aan ongeveer 10MB

restapi:
    listen: 172.16.0.22:8008
    connect_address: 172.16.0.22:8008

consul:
    host: 172.16.0.11:8500

bootstrap:
    dcs:
        ttl: 30
        loop_wait: 10
        retry_timeout: 10
        maximum_lag_on_failover: 1048576 #staat gelijk aan ongeveer 1MB
        master_start_timeout: 300
        postgresql:
            use_pg_rewind: true
            parameters:
                archive_command: 'exit 0'
                archive_mode: 'on'
                autovacuum: 'on'
                checkpoint_completion_target: 0.6
                checkpoint_warning: 300
                datestyle: 'iso, mdy'
                default_text_search_config: 'pg_catalog.english'
                effective_cache_size: '128MB'
                hot_standby: 'on'
                include_if_exists: 'repmgr_lib.conf'
                lc_messages: 'C'
                listen_addresses: '*'
                log_autovacuum_min_duration: 0
                log_checkpoints: 'on'
```

```
                logging_collector: 'on'
                log_min_messages: INFO
                log_filename: 'postgresql.log'
                log_connections: 'on'
                log_directory: '/var/log/postgresql'
                log_disconnections: 'on'
                log_line_prefix: '%t [%p]: [%l-1] user=%u,db=%d,app=%a '
                log_lock_waits: 'on'
                log_min_duration_statement: 0
                log_temp_files: 0
                maintenance_work_mem: '128MB'
                max_connections: 101
                max_wal_senders: 5
                port: 5432
                shared_buffers: '128MB'
                shared_preload_libraries: 'pg_stat_statements'
                unix_socket_directories: '/var/run/postgresql'
                wal_buffers: '8MB'
                wal_keep_segments: '200'
                wal_level: 'replica'
                work_mem: '128MB'

    initdb:
    - encoding: UTF8
    - data-checksums

    pg_hba:
    - host replication replicator 127.0.0.1/32 md5
    - host replication replicator 172.16.0.22/0 md5
    - host replication replicator 172.16.0.33/0 md5
    - host all postgres 172.16.0.11/32 trust
    - host all postgres 172.16.0.22/32 trust
    - host all postgres 172.16.0.33/32 trust
    - host all all 0.0.0.0/0 md5
    - local all all peer

    users:
        admin:
            password: admin
            options:
                - createrole
                - createdb

postgresql:
    listen: 127.0.0.1,172.16.0.22:5432
    connect_address: 172.16.0.22:5432
    data_dir: /var/lib/postgresql/patroni
    pgpass: /tmp/pgpass
    use_unix_socket: true
```

```
    authentication:
        replication:
            username: replication
            password: replication
        superuser:
            username: postgres
            password: postgres
        parameters:
            unix_socket_directories: '/var/run/postgresql'
            wal_compression: on

tags:
    nofailover: false
    noloadbalance: false
    clonefrom: false
    nosync: false

watchdog:
    mode: off
```

ttl geeft de tijd, in seconden, mee voor de primary node om het leader lock te verwerven. Wanneer deze ttl gepasseerd is, zal failover plaatsvinden. Standaard staat deze waarde op 30 seconden.

Bij retry_timeout wordt de tijd meegegeven die na overschrijden, de primary node zal degraderen, in geval van time-out. Standaard staat deze waarde op 10 seconden.

maximum_lag_on_failover geeft het maximaal aantal bytes mee dat een standby node mag vertragen (lag) om deel te nemen aan een verkiezing van nieuwe primary node, in geval van failover.

Bij master_start_timeout wordt de tijd meegegeven waarop een primary node de tijd krijgt om te herstellen van een fout voordat failover ingeschakeld wordt. Standaard staat deze parameter op 300 seconden. Als deze parameter op 0 staat, zal direct failover worden gedaan.

use_pg_rewind zal hier op true staan, omdat deze optie gebruikt zal worden. Standaard staat dit op false.

Bij parameters is er een lijst terug te vinden van commando's die gebruikt worden om Postgres te configureren.

pg_hba zorgt voor client authenticatie tussen de PostgreSQL server en de client applicatie.

Bij tags zal nofailover aan de hand van de meegegeven parameter kijken of de node mag deelnemen aan de verkiezing om nieuwe primary node te worden. clonefrom zal dienen, wanneer true, voor de andere nodes om deze node te gebruiken voor bootstrap. nosync zal indien true nooit geselecteerd worden voor synchrone replica.

Hierna zal opnieuw voor Consul een service toegevoegd worden die bij het opstarten van de server wordt uitgevoerd. Belangrijk is dat bij het aanmaken van deze service, de service gestart en ge-enabled moet zijn. In deze service wordt verwezen naar het config bestand voor de Consul agent (cliënt). Deze ziet er als volgt uit.

```
{
    "server": false,
    "datacenter": "dc1",
    "data_dir": "/var/consul/client",
    "ui_dir": "/var/consul/ui",
    "encrypt": "/q/vkVS+My2nl8Zk/8csuQ==",
    "log_level": "INFO",
    "enable_syslog": true,
    "start_join": ["172.16.0.11"],
    "bind_addr": "172.16.0.22"
}
```

9.5 pgNode2

Tijdens de configuratie van de cluster zal pgNode2 '172.16.0.33' als ip adres hebben.

Idem als bij pgServer is de eerste stap het installeren van de juiste packages. Hier gaat het over Python, Consul Agent, Patroni en PostgreSQL uiteraard. Dit zal opnieuw gebeuren aan de hand van het postgresql-cluster-setup.sh bestand die wordt aangeroepen in de Vagrantfile.

Bij het configureren van Patroni zal verwezen worden naar pgNode2Patroni.yml waarin de configuratie te vinden is. Dit bestand is zeer gelijkend aan wat er bij pgNode1 staat, maar kent toch een paar kleine wijzigingen, maar enkel in ip-adressen:

```
restapi:
    listen: 172.16.0.33:8008
    connect_address: 172.16.0.33:8008

consul:
    host: 172.16.0.11:8500
```

```
postgresql:
    listen: 127.0.0.1,172.16.0.33:5432
    connect_address: 172.16.0.33:5432
    data_dir: /var/lib/postgresql/patroni
    pgpass: /tmp/pgpass
    use_unix_socket: true
    authentication:
        replication:
            username: replication
```

```
        password: replication
    superuser:
        username: postgres
        password: postgres
parameters:
    unix_socket_directories: '/var/run/postgresql'
    wal_compression: on
```

Hierna zal opnieuw voor Consul een service toegevoegd worden die bij het opstarten van de server wordt uitgevoerd. Belangrijk is dat bij het aanmaken van deze service, de service gestart en ge-enabled moet zijn. In deze service wordt verwezen naar het config bestand voor de Consul agent (cliënt). Deze ziet er als volgt uit.

```
{
    "server": false,
    "datacenter": "dc1",
    "data_dir": "/var/consul/client",
    "ui_dir": "/var/consul/ui",
    "encrypt": "/q/vkVS+My2nl8Zk/8csuQ==",
    "log_level": "INFO",
    "enable_syslog": true,
    "start_join": ["172.16.0.11"],
    "bind_addr": "172.16.0.33"
}
```

In onderstaande blok code staat het postgresql-setup-cluster.sh bestand die door Vagrant zal aangeroepen worden. Hierin zullen de twee nodes en server mee geconfigureerd worden.

```
#!/bin/bash

POSTGRESQL_VERSION=12
PGBOUNCER_VERSION=1.15.0-1.pgdg16.04+1
CONSUL_VERSION=1.9.5
PATRONI_VERSION=2.0.2
PSYCOPG2_VERSION=2.8.6
PYCONSUL_VERSION=1.2.3

PG_PATH="/etc/postgresql/${POSTGRESQL_VERSION}/main"

function setup_packages() {
    apt-get update
    apt-get -y install wget unzip curl libpq-dev ca-certificates ntp
        tree
}
```

```
function setup_python() {
    #installl pip
    apt-get -y install python python-pip
}

function setup_patroni() {
    # Install
    pip install -U pip
    pip install psycopg2-binary==${PSYCOPG2_VERSION}
    pip install python-consul==${PYCONSUL_VERSION}
    pip install -U flake8 configparser zipp
    pip install patroni[consul]==${PATRONI_VERSION}
    mkdir -p /etc/patroni
    mkdir -p /var/lib/postgresql/patroni
    chmod 700 /var/lib/postgresql/patroni
    mkdir -p /var/log/patroni
    chmod 777 /var/log/patroni

    # Configure Patroni on nodes
    if [ "$(hostname)" == "pgNode1" ]; then
    cp -p /vagrant/pgNode1Patroni01.yml /etc/patroni/patroni.yml
    elif [ "$(hostname)" == "pgNode2" ]; then # consul server and
        client
    cp -p /vagrant/pgNode1Patroni02.yml /etc/patroni/patroni.yml
    fi
    cp -p /vagrant/patroni.service /etc/systemd/system/
    echo "export PATRONICTL_CONFIG_FILE=/etc/patroni/patroni.yml" >>
        /etc/environment
    source /etc/environment
    systemctl daemon-reload
}

function setup_consul() {
    # Install consul on 2 nodes en 1 server
    curl -s -O
        https://releases.hashicorp.com/consul/${CONSUL_VERSION}/consul_${CONSUL_VERSION
    unzip consul_${CONSUL_VERSION}_linux_amd64.zip
    mv consul /usr/local/bin/
    rm -f consul_${CONSUL_VERSION}_linux_amd64.zip
    mkdir -p /etc/consul.d/{server,client}

    if [ "$(hostname)" == "pgNode1" ]; then
    cp -p /vagrant/consul.d/client/pgNode1.json
        /etc/consul.d/client/pgNode1.json
    fi
    if [ "$(hostname)" == "pgNode2" ]; then
    cp -p /vagrant/consul.d/client/pgNode2.json
        /etc/consul.d/client/pgNode2.json
    fi
```

```
    if [ "$(hostname)" == "pgServer" ]; then
    cp -rp /vagrant/consul.d/server/ /etc/consul.d
    fi
    mkdir -p /var/consul/ui
    useradd -M -d /var/consul -s /bin/bash consul

    # Configure consul service
    mkdir -p /var/consul/{server,client}
    if [ "$(hostname)" == "pgNode1" ]; then # consul client
    cp -p /vagrant/consul-client.service /etc/systemd/system/
    fi
    if [ "$(hostname)" == "pgNode2" ]; then # consul client
    cp -p /vagrant/consul-client.service /etc/systemd/system/
    fi
    if [ "$(hostname)" == "pgServer" ]; then # consul server
    cp -p /vagrant/consul-server.service /etc/systemd/system/
    fi
    systemctl daemon-reload

    chown -R consul:consul /var/consul/

    #start consul service
    if [ "$(hostname)" == "pgNode1" ]; then
    systemctl start consul-client
    systemctl enable consul-client
    fi
    if [ "$(hostname)" == "pgNode2" ]; then
    systemctl start consul-client
    systemctl enable consul-client
    fi
    if [ "$(hostname)" == "pgServer" ]; then
    systemctl start consul-server
    systemctl enable consul-server
    fi
}

function setup_haproxy() {
    # Install haproxy
    apt-get install haproxy -y

    # Configure
    cp -p /vagrant/haproxy.cfg /etc/haproxy/haproxy.cfg

    # start haproxy.service
    systemctl restart haproxy
    systemctl enable haproxy
}

function setup_pgbouncer() {
```

```
    # Install psycopg2 zodat python scripts kunnen runnen
    pip install psycopg2-binary==${PSYCOPG2_VERSION}

    # Install the version that comes with the official apt postgresql
       repository
    apt-get -y install pgbouncer=${PGBOUNCER_VERSION}
       postgresql-client-${POSTGRESQL_VERSION}

    cat > /etc/pgbouncer/pgbouncer.ini <<EOF
    [databases]
    postgres = host=172.16.0.11 port=5000 pool_size=6
    template1 = host=172.16.0.11 port=5000 pool_size=6
    test = host=172.16.0.11 port=5000 pool_size=6

    [pgbouncer]
    logfile = /var/log/postgresql/pgbouncer.log
    pidfile = /var/run/postgresql/pgbouncer.pid
    listen_addr = *
    listen_port = 6432
    unix_socket_dir = /var/run/postgresql
    auth_type = trust
    auth_file = /etc/pgbouncer/userlist.txt
    admin_users = postgres
    stats_users =
    pool_mode = transaction
    server_reset_query =
    server_check_query = select 1
    server_check_delay = 10
    max_client_conn = 1000
    default_pool_size = 12
    reserve_pool_size = 5
    log_connections = 1
    log_disconnections = 1
    log_pooler_errors = 1
    EOF

    cat > /etc/pgbouncer/userlist.txt <<EOF
    "postgres" "password"
    EOF

    cat > /etc/default/pgbouncer <<EOF
    START=1
    EOF
    systemctl restart pgbouncer
    systemctl enable pgbouncer
}

function setup_postgresql_repo() {
    # Setup postgresql repo
```

```
    echo "deb http://apt.postgresql.org/pub/repos/apt/ $(lsb_release
        -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list

    # Setup postgresql repo key
    wget --quiet -O -
        https://www.postgresql.org/media/keys/ACCC4CF8.asc | apt-key
        add -

    apt-get update
}

function setup_postgresql() {
    # Install postgresql
    apt-get -y install postgresql-${POSTGRESQL_VERSION}

    #Stop service
    systemctl stop postgresql
    systemctl disable postgresql

    chown -R postgres:postgres /var/lib/postgresql/patroni
    chown -R postgres:postgres /etc/patroni


    # Patroni binaries check
    ln -f -s /usr/lib/postgresql/${POSTGRESQL_VERSION}/bin/ * /usr/bin/

    # Start patroni.service
    systemctl start patroni
    systemctl enable patroni
}

# Timezone
timedatectl set-timezone Europe/Brussels

setup_packages

if [ ! -f /usr/bin/pip ]; then
setup_python
fi

if [ "$(hostname)" != "pgServer" ]; then
if [ ! -f /usr/local/bin/patroni ]; then
setup_patroni
fi
fi

if [ ! -f /usr/local/bin/consul ]; then
setup_consul
fi
```

```
# Install HAProxy in pgServer
if [ "$(hostname)" == "pgServer" ]; then
if [ ! -f /etc/haproxy/haproxy.cfg ]; then
setup_haproxy
fi
fi

if [ ! -f /etc/apt/sources.list.d/pgdg.list ]; then
setup_postgresql_repo
fi

if [ "$(hostname)" != "pgServer" ]; then
if [ ! -f ${PG_PATH}/postgresql.conf ]; then
setup_postgresql
fi
fi

if [ "$(hostname)" == "pgServer" ]; then
if [ ! -f /etc/pgbouncer/pgbouncer.ini ]; then
setup_pgbouncer
fi
fi
```

Deze proof of concept is gebaseerd op een cluster gemaakt door Vicenç Juan Tomàs Montserrat Montserrat (2020). Zijn opstelling is de basis geweest voor het opzetten van deze Patroni cluster. De opzet van deze cluster omvatten de drie functionaliteiten die nodig waren om te voldoen aan de functionele requirements, namelijk replicatie, monitoring en failover.

9.6 Persoonlijke conclusie

Het opzetten van de Patroni cluster verliep voor mij redelijk moeizaam. Het was een proces van trial and error. Het was niet direct duidelijk welke configuraties en tools er precies nodig waren voor de opzet van de cluster. Wat mij wel direct opviel, is dat alles te configureren valt in de verschillende configuratiebestanden. De cluster kan naar believen geconfigureerd worden volgens de verschillende noden van gebruikers.

10. Conclusie

In dit onderzoek wordt een antwoord gegeven op de onderzoeksvraag: Welke PostgreSQL High Availability cluster oplossing kunnen bedrijven gebruiken om garantie te hebben op monitoring, redundantie en failover? Om hierop een antwoord te vinden, is een vergelijkende studie uitgevoerd tussen de verschillende oplossingen uit dit onderzoek. Hierbij werd elke oplossing afgetoetst aan de requirements uit Hoofdstuk 3.

Uit de resultaten van de requirementsanalyse blijkt dat Patroni momenteel de PostgreSQL High Availability cluster oplossing is die het meest aansluit aan de vooropgestelde requirements. De proof of concept beaamt ook deze vaststelling. Het gebruik van Patroni geeft een garantie op High Availability in een PostgreSQL cluster. Patroni zelf is op het vlak van replicatie zeer flexibel. Het werkt standaard met asynchrone replicatie, maar kan ook zo geconfigureerd worden dat het synchroon gebeurt. Wat betreft failover is Patroni een oplossing met een uitgebreide configuratie. Er zijn veel geavanceerde opties voor het configureren van failover in de cluster. Een nuttige optie is dat failover automatisch kan gebeuren en dus geen nood hoeft te hebben aan manuele interventies. Inzake monitoring kent Patroni een zeer rijke REST API die kan gebruikt worden om de cluster en de nodes te monitoren. Hierin kunnen persoonlijke voorkeuren van monitoring toegevoegd worden.

Ook PostgreSQL Automatic Failover (PAF) is een goed alternatief bij het opzetten van een PostgreSQL High Availability cluster. Het voldoet voldoende aan de verschillende (functionele) requirements om als alternatief gezien te worden van Patroni als oplossing. Aan de hand van de tool Pacemaker kan PostgreSQL Automatic Failover (PAF) zodanig geconfigureerd worden dat er failover en monitoring gedaan kan worden.

De overige twee tools: Pgpool-II en Replication Manager (repmgr) zijn zeker ook waardevolle oplossingen bij het opzetten van PostgreSQL High Availability cluster. Echter

voldoen deze twee oplossingen minder aan de requirements zoals Patroni en PostgreSQL Automatic Failover (PAF).

Bij de aanvang van dit onderzoek werden deze resultaten niet verwacht. Dit omdat de verschillende oplossingen nog onvoldoende aan bod waren gekomen om de resultaten te voorspellen. Het was pas na het lezen van het artikel van Akhtar, H: PostgreSQL High Availability: The Considerations and Candidates dat er een idee gevormd werd van welke PostgreSQL oplossingen er allemaal aanwezig waren. Akhtar haalde Patroni, PostgreSQL Automatic Failover (PAF), pgPool-II en Replication Manager (repmgr) aan als gepaste oplossingen voor High Availability in een PostgreSQL omgeving. Uit de literatuurstudie bleek dat Patroni een erg sterke PostgreSQL High Availability oplossing bood. Na verdere verdieping en aan de hand van de requirementanalyse uit Hoofdstuk 4 werden deze assumpties over Patroni bevestigd en beargumenteerd. Dit onderzoek kan een basis vormen voor bedrijven die zich willen inwerken in PostgreSQL High Availability. Er wordt hier een duidelijk beeld gegeven van de verschillende oplossingen en de manier waarop deze oplossingen voldoen aan bepaalde requirements. Deze requirements zijn gekomen uit de bedrijfswereld en voldoen hierbij aan de noden van andere bedrijven.

Dit onderzoek vraagt zeker naar een vervolgonderzoek voor de toekomst. Hoe wordt High Availability binnen tien jaar geïmplementeerd in PostgreSQL clusters? Welke nieuwe noden in een cluster zijn er dan van belang? Vervolgonderzoek zal zeker een meerwaarde bieden aan bedrijven die belang hechten aan de duurzaamheid van hun clusters.

A. Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Samenvatting

In deze bachelorproef zal onderzoek gedaan worden naar de huidige staat van open-source database high availability (HA) tooling. De vraag naar PostgreSQL (pgSQL) wordt steeds groter, waardoor een onderzoek naar open-source database high availability (HA) tooling zeker niet onmisbaar is. In het eerste deel van dit onderzoek zal er geduid worden wat open-source database high availability (HA) tooling precies is en wat de toepassingen ervan zijn. Hierna wordt een vergelijkende studie uitgevoerd over de verschillende open-source database high availability (HA) oplossingen. Na de vergelijkende studie wordt als proof-of-concept een PostgreSQL (pgSQL) cluster opgezet. In dit onderzoek zal ik worden bijgestaan door Ruben Demey, Global IT Operations Manager bij ST Engineering iDirect. Het verwachte resultaat van dit onderzoek is aantonen dat er een PostgreSQL high availability (ha) cluster oplossing aanwezig is die voldoet aan de requirements, opgesteld in dit onderzoek. In de toekomst is het interessant om soortgelijk onderzoek uit te voeren, om te kijken of de tools die in dit onderzoek worden aangehaald, nog steeds even waardevol zullen zijn.

A.2 Introductie

Database high availability (HA) staat voor de garantie van het behouden van gegevens in geval er zich een defect of storing voordoet aan de databank server. Een storing aan een databank server kan te wijten zijn aan verschillende factoren. Voorbeelden hiervan zijn het verlies van netwerkconnectie en een defect in de software of hardware van de databankserver. Ook menselijke factoren en omgevingsfactoren moeten in rekening genomen worden. Voorbeelden hiervan zijn een menselijke vergissing en een wijziging in temperatuur. Investeren in high availability geeft meer zekerheid over de beschikbaarheid van data en biedt verschillende mogelijkheden voor failover en systeembescherming (IBM, 2019). Met behulp van clusters kan er één actieve en een of meerdere standby instanties van de databank server zijn. Een cluster is een groep van servers en computers die samenwerken met elkaar alsof het één systeem is. Deze standby instanties zullen, in het ideale geval, dezelfde gegevens bevatten als de actieve server (Bermingham, 2019). Wanneer dan een actieve server faalt, kan een standby instantie inspringen waardoor dataverlies en server downtime gereduceerd worden.

A.3 State of the art

Singer spreekt over high availability (HA) clustering als een groep van servers die applicaties en services ondersteunen die op een betrouwbare manier gebruikt kunnen worden met een minimaal aantal aantal downtimes. Hij bespreekt de cluster architectuur en wat de best practice is voor high availability (HA) binnen een cluster. De conclusie die hier getrokken wordt, is dat het primaire doel van een high availability (HA) systeem het voorkomen en elimineren van alle single points of failure zijn. Dit systeem moet beschikken over meerdere geteste actieplannen. Dit zodat ze in geval van storing, verstoring en defect in dienstverlening direct, gepast en onafhankelijk kunnen reageren. Zorgvuldige planning + betrouwbare implementatiemethoden + stabiele softwareplatforms + degelijke hardware-infrastructuur + vlotte technische operaties + voorzichtige managementdoelstellingen + consistente databeveiliging + voorspelbare redundantiesystemen + robuuste backupoplossingen + meerdere herstelopties = 100% uptime (Singer, 2020). Ook Jevtic heeft het over veel van de punten die hierboven reeds zijn aangehaald. Jevtic spreekt over een highly available architectuur waarin meerdere componenten samenwerken om een ononderbroken service gedurende een bepaalde periode te garanderen. Dit omvat ook de reactietijd op verzoeken van gebruikers. Jevtic kenmerkt een highly available (HA) infrastructuur aan de hand van: 1. Hardware redundantie; 2. Software en applicatie redundantie; 3. Gegevens redundantie; 4. Elimineren van storingspunten (Jevtic, 2018). Akhtar heeft vier van de meest gebruikte database high availability (HA) oplossingen opgelijst. Deze vier zijn "PgPool-II", "PostgreSQL Automatic Failover (PAF)", "RepMgr [Replication Manager] ", en "Patroni"(Akhtar, 2020). Akhtar vergelijkt deze verschillende oplossingen kort met elkaar. Akhtar definieert high availability (HA) als niet alleen de continuïteit van een bepaalde service, maar volgens hem gaat high availability (HA) ook over het vermogen van een systeem om een (hogere) werkdruk te kunnen schalen en te beheren. Dit systeem moet volgens Akhtar de gemiddelde werkdruk, maar ook de piekmomenten aankunnen.

Aldus Andersen zijn de top drie open-source databanken van 2019, in volgorde van top 3, MySQL met 31.7%, PostgreSQL met 13.4% en MongoDB met 12.2% (Anderson, 2020) van het totaal aantal open-source databank gebruikers.

A.4 Methodologie

In de eerste fase van het onderzoek zal er een vergelijkende studie gebeuren over de huidige, database high availability (HA) oplossingen. Deze verschillende tools/oplossingen zullen dan met elkaar vergeleken worden. Hierbij wordt gekeken naar welke elementen er allemaal (meermaals) voorkomen. Hiervan komt er een lijst die gebruikt zal worden om te schiften tussen de verschillende oplossingen. Op deze manier zal er dan een oplossing gekozen worden. Via deze methode wordt er gekeken om maximum 3 verschillende oplossingen uit te werken, waarbij één oplossing gebruikt zal worden bij de proof of concept. In de tweede fase van het onderzoek wordt de focus gelegd op het opzetten van de PostgreSQL (pgSQL) cluster als proof of concept. PostgreSQL is een open-source, object-relationeel databank systeem (PostgreSQL, 2020). Bij Inuits, een Belgisch open-source bedrijf met verschillende vestigingen in Europa, merken ze een stijging in de vraag naar het PostgreSQL verhaal. Deze zal vooraf gegaan worden door een literatuurstudie over PostgreSQL (pgSQL). Aan de hand hiervan zal er gewerkt worden aan het opbouwen van de PostgreSQL (pgSQL) cluster. Vooraleer dit geautomatiseerd wordt, zal de opbouw manueel verlopen. De opbouw zal gebeuren via virtuele machines (VirtualBox) waarop Linux-distributies staan. In het onderzoek zal dus gebruik gemaakt worden van Linux-servers. De keuze van Linux-distributie zal onderbouwd worden in dit onderzoek. De opbouw van de cluster zal telkens grondig gedocumenteerd worden. Na het opzetten van de PostgreSQL (pgSQL) cluster zal er getest worden of de gebruikte database high availability (HA) oplossing functioneel is. Deze testen zullen uitgebreid beschreven worden.

A.5 Verwachte resultaten

Uit het onderzoek zal blijken dat verschillende database high availability (HA) oplossingen mogelijk zijn binnen een PostgreSQL (pgSQL) cluster. De opbouw van deze cluster zal gebeuren aan de hand van virtuele machines. Wanneer de virtuele machine, waarop de PostgreSQL server staat, uitvalt, zal er een standby instantie van deze server het werk van de actieve, uitgevallen server overnemen. Hierdoor zal er geen downtime of dataverlies zijn. De data zal beschikbaar en onverstoord blijven.

Uit dit onderzoek zullen ook best practices volgen die gehanteerd kunnen worden om op die manier het risico op verlies van gegevens te verminderen. De kans om offline te zijn zal lager liggen met een high available systeem.

A.6 Verwachte conclusies

Uit het onderzoek zal blijken dat database high availability (HA) een blijvend topic is waar voldoende aandacht aan besteed moet worden in kleine en grote bedrijven. Zonder de implementatie van een high available (HA) architectuur kan een storing of downtime van de databank (SQL) server grote gevolgen hebben op een bedrijf/organisatie. Gevolgen zoals verlies van vertrouwen bij klanten, verlies van inkomen, verlies van informatie. Door middel van hardware-, software- en gegevensredundantie en het elimineren van mogelijke storingspunten zal er high availability gegarandeerd worden. De kost en tijd die geïnvesteerd moet worden in het onderhouden van een high available systeem zal lager liggen dan de kost en tijd die geïnvesteerd moet worden in geval van een downtime. Met de proof of concept wordt dan aangetoond dat database high availability (HA) eenvoudig te implementeren valt in een PostgreSQL (pgSQL) cluster.

Bibliografie

Ahmad, K. S., Ahmad, N., Tahir, H. & Khan, S. (2017). Fuzzy$_{M}$oSCoW : A fuzzy based MoSCoW method for *2017 International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*.

Akhtar, H. (2020, augustus 10). *PostgreSQL High Availability: The Considerations and Candidates*. Verkregen 2 januari 2021, van https://www.highgo.ca/2020/08/10/postgresql-high-availability-the-considerations-and-candidates/

Anderson, K. (2020, januari 17). *2019 Open Source Database Report*. DZone. Verkregen 31 december 2020, van https://dzone.com/articles/2019-open-source-database-report-top-databases-pub

Augustine, J. (2019, juli 10). *PostgreSQL WAL Retention and Clean Up: pg$_{a}$rchivecleanup*. Percona. Verkregen 13 april 2021, van https://www.percona.com/blog/2019/07/10/wal-retention-and-clean-up-pg_archivecleanup/

AVINetworks. (2020, september 26). *Failover Definition*. AVI Networks. Verkregen 1 april 2021, van https://avinetworks.com/glossary/failover/

Barman. (2020a). *Barman*. Barman. Verkregen 13 april 2021, van https://www.pgbarman.org/

Barman. (2020b). *Bringing you the Espresso Backup! About Barman*. Barman. Verkregen 13 april 2021, van https://www.pgbarman.org/about/

Bermingham, D. (2019, mei 9). *Clustering for SQL Server High Availability*. Big Data Quarterly (BDQ). Verkregen 31 december 2020, van https://www.dbta.com/BigDataQuarterly/Articles/Clustering-for-SQL-Server-High-Availability-131639.aspx

Carchedi, N. (2020, oktober 23). *What is SQL?* https://www.datacamp.com/community/tutorials/what-is-sql?utm_source=adwords_ppc&utm_campaignid=898687156&utm_adgroupid=48947256715&utm_device=c&utm_keyword=&utm_matchtype=b&utm_network=g&utm_adpostion=&utm_creative=229765585183&utm_

targetid=dsa-429603003980&utm_loc_interest_ms=&utm_loc_physical_ms=1001208&gclid=Cj0KCQjwse-DBhC7ARIsAI8YcWJOxNwWfRDfjloPnAcswV6QsafwgHI0hqagQFH5fj59JLPM04NOxMaAmudEALw_wcB

CDNF. (2020, juli 22). *What is a CNF?* cdnf. Verkregen 8 april 2021, van https://cdnf.io/what_is_cnf/

Codecademy. (2018). What is a Relational Database Management System? Learn about RDBMS and the language used to access large datasets SQL. https://www.codecademy.com/articles/what-is-rdbms-sql#:~:text=A%5C%20relational%5C%20database%5C%20is%5C%20a,database%5C%20is%5C%20organized%5C%20into%5C%20tables

CriticalCase. (2020). *5 reasons why you need high-availability for your business: What does it mean High-Availability infrastructure (HA)?* CriticalCase. Verkregen 12 mei 2021, van https://www.criticalcase.com/blog/5-reasons-why-you-need-high-availability-for-your-business.html

CSharp-Corner. (2019, september 6). *What Are Object-Oriented Databases And Their Advantages*. https://www.c-sharpcorner.com/article/what-are-object-oriented-databases-and-their-advantages2/#:~:text=An%20object-oriented%20database%20(OODBMS,object-oriented%20database%20management%20systems

DB-Engines. (2021a). *DB-Engines Ranking*. DB-Engines. Verkregen 25 maart 2021, van https://db-engines.com/en/ranking

DB-Engines. (2021b). *PostgreSQL System Properties*. DB-Engines. Verkregen 25 maart 2021, van https://db-engines.com/en/system/PostgreSQL

de Rorthais, M. R. J.-G. (2020). *PostgreSQL Automatic Failover*. clusterlabs. Verkregen 15 april 2021, van https://clusterlabs.github.io/PAF/

IBM. (2019). *High availability for databases*. International Business Machines Corporation (IBM). Verkregen 31 december 2020, van https://www.ibm.com/support/knowledgecenter/SSANHD_7.6.1.2/com.ibm.mbs.doc/gp_highavail/c_ctr_ha_for_databases.html

Ihalainen, A. V. F. L. C. J. A. N. (2019, april 4). *Replication Between PostgreSQL Versions Using Logical Replication*. Percona. Verkregen 15 april 2021, van https://www.percona.com/blog/2019/04/04/replication-between-postgresql-versions-using-logical-replication/#:~:text=Logical%20replication%20in%20PostgreSQL%20allows,is%20not%20open%20for%20writes.

Insausti, S. (2019, juli 18). *Scaling PostgreSQL for Large Amounts of Data*. Several Nines. Verkregen 1 april 2021, van https://severalnines.com/database-blog/scaling-postgresql-large-amounts-data

Jevtic, G. (2018, juni 22). *What is High Availability Architecture? Why is it Important?* Verkregen 25 maart 2021, van https://phoenixnap.com/blog/what-is-high-availability

Kalow, B. (2020). *Introduction to Vagrant*. HashiCorp. Verkregen 9 mei 2021, van https://www.vagrantup.com/intro

Kumar, V. (2020, april 7). *What Does "Database High Availability"Really Mean?* EDB. Verkregen 25 maart 2021, van https://www.enterprisedb.com/blog/what-does-database-high-availability-really-mean

Lutkevich, B. (2021). *high availability (HA): What is high availability?* (A. S. Gillis, Red.). TechTarget. Verkregen 25 maart 2021, van https://searchdatacenter.techtarget.com/definition/high-availability

Markwort, J. (2018, september 27). *Patroni: Setting up a highly available PostgreSQL cluster*. Cybertec. Verkregen 8 april 2021, van https://www.cybertec-postgresql.com/en/patroni-setting-up-a-highly-available-postgresql-cluster/

Montserrat, V. J. T. (2020, juni 5). *postgresql-patroni-setup*. Verkregen 27 april 2021, van https://github.com/vtomasr5/postgresql-patroni-setup

MySQL. (2021). *1.5 Point-in-Time (Incremental) Recovery*. MySQL. Verkregen 22 april 2021, van https://dev.mysql.com/doc/mysql-backup-excerpt/8.0/en/point-in-time-recovery.html#:~:text=Point-in-time%20recovery%20refers,time%20the%20backup%20was%20made

Nethosting. (2019, mei 13). *MySQL vs. PostgreSQL: 2019 Showdown*. https://nethosting.com/mysql-vs-postgresql-2019-showdown/

Oracle. (2021). *Database defined*. Oracle. Verkregen 15 april 2021, van https://www.oracle.com/database/what-is-database/

pgpool. (2021). *What is Pgpool-II?* pgpool. Verkregen 15 april 2021, van https://www.pgpool.net/mediawiki/index.php/Main_Page

PostgreSQL. (2020). PostgreSQL. Verkregen 31 december 2020, van https://www.postgresql.org/

PostgreSQL. (2021a). *About: What is PostgreSQL?* PostgreSQL. Verkregen 25 maart 2021, van https://www.postgresql.org/about/

PostgreSQL. (2021b). *pg_basebackup*. PostgreSQL. Verkregen 22 april 2021, van https://www.postgresql.org/docs/10/app-pgbasebackup.html

PostgreSQL. (2021c). *pg_rewind*. PostgreSQL. Verkregen 15 april 2021, van https://www.postgresql.org/docs/12/app-pgrewind.html

PostgreSQL. (2021d). *What is Postgres?* PostgreSQL. Verkregen 25 maart 2021, van https://www.postgresql.org/docs/6.3/c0101.htm

PostgreSQL. (2021e). *Write-Ahead Logging (WAL)*. PostgreSQL. Verkregen 12 mei 2021, van https://www.postgresql.org/docs/9.0/wal-intro.html

postgresql. (2021). *PostgreSQL: The World's Most Advanced Open Source Relational Database*. PostgreSQL. Verkregen 25 maart 2021, van https://www.postgresql.org/

Raima. (2021). *High-Availability Database (HA DB)*. Raima. Verkregen 20 april 2021, van https://raima.com/rdme-high-availability-database/

Ramachandran, M. (2019, maart 19). *PostgreSQL Connection Pooling With PgBouncer: Learn more about using PgBouncer to pool PostgreSQL connections.* Database Zone. Verkregen 12 mei 2021, van https://dzone.com/articles/postgresql-connection-pooling-with-pgbouncer#:~:text=PgBouncer%20is%20an%20open-source,each%20unique%20user,%20database%20pair.

Ranganathan, S. (2020, december 9). *Logische decodering: Logische decodering starten*. Microsoft. Verkregen 25 mei 2021, van https://docs.microsoft.com/nl-nl/azure/postgresql/concepts-logical

repmgr. (2021a). *repmgr standby switchover*. repmgr. Verkregen 13 april 2021, van https://repmgr.org/docs/4.0/repmgr-standby-switchover.html

repmgr. (2021b). *What is repmgr?* repmgr. Verkregen 15 april 2021, van https://repmgr.org/

Roth, P. M. M. R. T. S. C. G. J. (2016, mei 17). *About Log Shipping (SQL Server)*. Microsoft. Verkregen 25 maart 2021, van https://docs.microsoft.com/en-us/sql/database-engine/log-shipping/about-log-shipping-sql-server?view=sql-server-ver15

ScaleGrid. (2018a, augustus 22). *Managing High Availability in PostgreSQL Part III: Patroni*. ScaleGrid. Verkregen 8 april 2021, van https://scalegrid.io/blog/managing-high-availability-in-postgresql-part-3/#:~:text=Patroni%20ensures%20the%20end-to,be%20customized%20to%20your%20needs.

ScaleGrid. (2018b, november 27). *Managing PostgreSQL High Availability Part I: PostgreSQL Automatic Failover*. ScaleGrid. Verkregen 15 april 2021, van https://scalegrid.io/blog/managing-high-availability-in-postgresql-part-1/

ServersAustralia. (2017, oktober 31). *What is High Availability (HA) and Do I Need It?* Servers Australia. Verkregen 25 maart 2021, van https://www.serversaustralia.com.au/resources/blog/what-is-high-availability-ha-and-do-i-need-it/#:~:text=The%20purpose%20of%20HA%20architecture,achieve%20maximum%20productivity%20and%20reliability

SeveralNines. (2020). *PostgreSQL Load Balancing with HAProxy: 2. What is HAProxy?* SeveralNines. Verkregen 12 mei 2021, van https://severalnines.com/resources/database-management-tutorials/postgresql-load-balancing-haproxy

Singer, D. (2020, augustus 6). *What is High Availability? A Tutorial*. Liquid Web. Verkregen 2 januari 2021, van https://www.liquidweb.com/kb/what-is-high-availability-a-tutorial/

Technopedia. (2021a). *High Availability (HA): What Does High Availability (HA) Mean?* Technopedia. Verkregen 25 maart 2021, van https://www.techopedia.com/definition/1021/high-availability-ha

Technopedia. (2021b, april 28). *Object-Relational Database (ORD)*. https://www.techopedia.com/definition/8714/object-relational-database-ord

TechTarget. (2017, november 16). *Cluster*. TechTarget. Verkregen 1 april 2021, van https://whatis.techtarget.com/definition/cluster

TechTarget. (2020). *Failback*. TechTarget. Verkregen 1 april 2021, van https://whatis.techtarget.com/definition/failback

wiki.postgresql. (2020, december 9). *Repmgr*. wiki.postgresql. Verkregen 13 april 2021, van https://wiki.postgresql.org/wiki/Repmgr#repmgr_5_Features

wiki.postgresql.org. (2020, juli 6). *Streaming Replication*. wiki.postgresql.org. Verkregen 8 april 2021, van https://wiki.postgresql.org/wiki/Streaming_Replication#:~:text=Streaming%20Replication%20(SR)%20provides%20the,some%20out%20of%20date%20information