



Destructieve query's detecteren

Afstudeerscriptie

Rick Bulthuis - 431868
Afstudeerbegeleider: Simon Paarhuis
Afstudeerdocent: Erik van der Arend

15 juni 2020
Versie 2.0

ONDERDEEL VAN
 **VISMA**

Inhoudsopgave

1. Versiebeheer	5
2. Samenvatting	7
3. Inleiding	8
4. De probleemstelling	9
5. Proces aanpak	10
5.1 Analyse	10
5.2 Ontwerp	10
5.3 Realisatie	10
6. Onderzoek	11
6.1 Transporteren query's	11
6.1.1 Onderscheppen van query's	11
6.1.1.1 MaxScale	11
6.1.1.2 Logging	12
6.1.1.3 DBconnector	12
6.1.1.4 Conclusie	12
6.1.2 Message brokers	12
6.1.2.1 RabbitMQ	13
6.1.2.1.1 Publisher	13
6.1.2.1.2 Queue	13
6.1.2.1.3 Consumer	13
6.1.2.1.4 Snelheid	14
6.1.2.1.5 Protocollen	14
6.1.2.2 Kafka	14
6.1.2.2.1 Publisher	14
6.1.2.2.2 Consumer	14
6.1.2.2.3 Snelheid	15
6.1.2.3 ZeroMQ	15
6.1.2.4 Conclusie	15
6.1.3 Queues	15
6.2 Query's analyseren	16
6.2.1 Explain (mysql/mariaDB)	16
6.2.2. Logging	17
6.3 Definitie destructieve query	17
6.4 Monitoren	18
6.5 Conclusie	19

7. Ontwerp	21
7.1 Requirements	21
7.2 Schets applicatie	22
7.3 Query detector	24
7.3.1 DBConnector	24
7.3.2 DBListener	24
7.3.3 ReceiveTopic	25
7.3.4 MariaDB - Query detector	25
7.4 Aanpassingen Verzuimsignaal applicatie	26
7.4.1 DBConnector	26
7.4.2 DBListener	26
7.5 Query detector manager	28
7.5.1 DBConnector	29
7.5.2 ReceiveTopic	29
7.4.3 Message	30
7.6 Output query resultaten	31
7.6.1 Aantal query's per actie	31
7.6.2 Aantal duplicaten	31
7.6.3 Doorlooptijd van een query	31
7.6.4 Rij waarde van een query	32
7.6.5 Explain bevat filesort of temporary tables	32
7.6.6 Conclusie	32
7.6.7 Graylog	33
8. Realisatie	34
8.1 verzuimsignaal applicatie	34
8.1.1 Opbouw bericht	34
8.1.2 Message bundels	35
8.1.3 Exchange declare of queue declare	35
8.2 RabbitMQ	36
8.3 Query detector manager	36
8.3.1 Consumeren berichten	36
8.1.2 Worker	37
8.4 MariaDB	37
8.4.1 Aantal query's per actie	37
8.4.2 Aantal duplicaten	38
8.4.3 doorlooptijd van een query	38
8.4.4 Rij waarde van een query	38
8.4.5 Explain bevat filesort of temporary tables	39
8.5 Graylog	39
8.6 Testresultaten	39

8.7 Resultaat	40
9. Slot	42
Literatuurlijst	43
Bijlagen	44
Bijlage 1: Onderzoeksverslag	44
Bijlage 2: Technisch Ontwerp	44

1. Versiebeheer

Versiebeheer			
Wanneer	Wat	Door Wie	
17-02-2020	Opzet onderzoeksrapport	Rick Bulthuis	0.1
07-03-2020	Start opzet Resultaat (verschillende soorten queues)	Rick Bulthuis	0.2
09-03-2020	Hoofdstuk verschillende soorten queues opgezet en aangevuld	Rick Bulthuis	0.3
17-03-2020	Start opzet message brokers	Rick Bulthuis	0.4
02-04-2020	Hoofdstuk kafka en zeromq uitgebreid	Rick Bulthuis	0.5
07-04-2020	Hoofdstuk query onderzoek explain opgezet/uitgebreid	Rick Bulthuis	0.6
12-04-2020	- Delen uit PvA toegevoegd - Onderzoek - Probleemstelling - Realisatie verder uitgeschreven	Rick Bulthuis	0.7
20-04-2020	Component diagram toegevoegd en uitgelegd	Rick Bulthuis	0.8
25-04-2020	Realisatie verder uitgeschreven	Rick Bulthuis	0.9
29-05-2020	- Klassediagrammen toegevoegd - Inleiding aangevuld - Monitoring toegevoegd	Rick Bulthuis	0.10
30-05-2020	Feedback	Erik van der Arend	0.11
01-06-2020	- Aanpak toegevoegd - Samenvatting geschreven - Slot geschreven - Onderzoek conclusie geschreven - Requirements toegevoegd - Testen toegevoegd - Literatuurlijst toegevoegd - Bijlagen Toegevoegd	Rick Bulthuis	0.12

02-06-2020	• Spellingscontrole • Definitief - concept versie	Rick Bulthuis	1.0
08-06-2020/ 14-06-2020	• Feedback verwerkt van Erik van den Arend en Edward Hertsenberg	Rick Bulthuis	1.1
15-06-2020	• Definitieve versie afstudeerscriptie	Rick Bulthuis	2.0

2. Samenvatting

In deze afstudeerscriptie wordt er een onderzoek gedaan hoe destructieve query's kunnen worden gedetecteerd. Een destructive query zorgt ervoor dat de database traag wordt of zelfs niet meer werkt. Met de resultaten van dit onderzoek en de wensen van de opdrachtgever is er een ontwerp gerealiseerd. Aan de hand van dit ontwerp is een prototype ontwikkeld.

Het probleem op dit moment is dat het soms voorkomt dat er query's zijn die niet goed functioneren. Een voorbeeld is dat een query een lange doorlooptijd heeft. Aangezien er een groot aantal query's over de lijn gaan, ongeveer 1 miljoen per seconde op piektijden, is het niet inzichtelijk welke query's dit zijn.

Als eerste is er onderzoek gedaan op het onderwerp hoe een query kan worden afgevangen. Uiteindelijk is er gekozen om dit op de DBConnector van de Verzuimsignaal applicatie te doen. Hier kan de benodigde data worden opgehaald en gaan alle query's van het systeem overheen. Vervolgens is er gekeken wat de beste manier was om de query's te transporteren naar een externe omgeving. Hier is uiteindelijk gekozen voor RabbitMQ. De keuze is hierop gevallen aangezien het genoeg data kan verplaatsen en het is al reeds aanwezig in de Verzuimsignaal applicatie.

Hiernaast is er bepaald op welke manier een query geanalyseerd zal worden. Hier is gekozen om "Explain" te gebruiken van MariaDB (Verzuimsignaal maakt gebruik van een mariaDB database). Dit geeft een goed inzicht in hoeveel stappen een query wordt uitgevoerd en of die bijvoorbeeld indexes gebruikt.

Verder is er gekeken wat de definitie van een query is. Het belangrijkste is de doorlooptijd van een query.

Tot slot is er gekeken wat er interessant is om te monitoren. Hier is ervoor gekozen om vooral totaalwaarden weer te geven. Denk bijvoorbeeld aan de totale uitvoertijd of het totaal aantal dubbele query's binnen een actie.

Vervolgens is er een ontwerp gerealiseerd voor de software. Hier is er rekening gehouden met een aantal requirements. De belangrijkste requirement is dat de Verzuimsignaal applicatie geen hinder mag ondervinden van het transporteren van de query's. Dit is gewaarborgd aan de hand van een berichten bundelaar. Ook zijn er geen onnodige acties uitgevoerd op de Verzuimsignaal applicatie.

Bij de realisatie is er aan de hand van het prototype naar boven gekomen dat er dubbele query's worden uitgevoerd binnen dezelfde actie. Ook is het inzichtelijk geworden welke query's een langere doorlooptijd hebben. Met deze gegevens worden destructieve query's naar voren gehaald.

3. Inleiding

Het bedrijf VerzuimSignaal B.V. richt zich op een passende oplossing voor het managen van het verzuim binnen verschillende organisaties. VerzuimSignaal B.V. helpt aan de hand van software, dat personeel duurzaam inzetbaar is en onnodig verzuim wordt voorkomen. De applicatie VerzuimSignaal heeft 2.5 miljoen werknemers geregistreerd. VerzuimSignaal B.V. kent drie applicaties. VerzuimSignaal is een van deze applicatie zoals hiervoor aangegeven. Ook is de applicatie DataExchange een onderdeel. Dit wordt ingezet als integratieplatform om gegevens uit te wisselen tussen derde partijen. Verder is er een applicatie genoemd Werknemer portaal. Dit portaal is specifiek ontwikkeld voor de werknemers die zijn geregistreerd in VerzuimSignaal. Hier kunnen werknemers inloggen en daar bekijken wat van verzuimverloop zij hebben.

Voor het bedrijf VerzuimSignaal B.V. zal er een tooling/applicatie ontwikkeld worden die zal dienen om destructive query's te detecteren. In deze afstudeerscriptie worden een aantal onderwerpen behandeld namelijk, wat is het probleem ([4. De probleemstelling](#)), het onderzoek op de hoofd en deelvragen ([6. Onderzoek](#)), verschillende ontwerpen van de softwareoplossingen ([7. Ontwerp](#)) en tot slot de realisatie van het product ([8. Realisatie](#)).

Bij dit onderzoek zal er worden ingezoomd op vier verschillende onderwerpen, namelijk:

- hoe kunnen de query's worden getransporteerd;
- hoe kunnen query's worden geanalyseerd;
- wat is de definitie van een destructive query;
- wat wordt er gemonitord.

Vervolgens zijn er requirements opgesteld en een aantal ontwerpen gemaakt van de software. Deze ontwerpen maken het inzichtelijk hoe de software in elkaar zit en hoe de software gaat werken tussen verschillende systemen.

Ten slotte zal in de realisatie de gemaakte keuzes in de software worden behandeld en uitgelegd hoe de software per component gaat werken.

4. De probleemstelling

Binnen de applicaties, Verzuimsignaal, DataExchange en Werknemer portaal is er momenteel nog geen oplossing om de destructieve query's van de databases te detecteren. Een destructive query zorgt ervoor dat de database traag wordt of zelfs niet meer werkt. Op het moment kan er alleen aan de hand van de server performers worden geconstateerd dat er problemen met de database zijn. Dit wil niet direct zeggen of er een probleem in de database is of dat het druk is op dat moment. Een ander kenmerk is dat het aantal meldingen op de supportafdeling erg kan oplopen. Maar op dat moment is het probleem al lopende. Vaak kunnen de problemen vooral gemerkt worden op de openstaande taken of dossiers. Deze lijsten kunnen soms wel 2.000 items bevatten. Wanneer de database niet soepel meer loopt kan dit gevolgen hebben op de applicaties.

Het probleem is dat er een grote hoeveelheid query's per seconde worden uitgevoerd. Dit kunnen op de piekmomenten wel meer dan 1 miljoen query's per seconde zijn. Deze query's zullen moeten worden getransporteerd om te analyseren.

Hoe het analyseren zal worden uitgevoerd is nog een vraagteken. Dit kan bijvoorbeeld aan de hand van een statische oplossing of met machine learning. Er moet ook worden gekeken of de database eventueel analyseer hulpmiddelen bevat. Maar tot dusver is het onduidelijk hoe het best de analyse wordt uitgevoerd.

Ook zal er nog duidelijk moeten worden wat de eigenschappen van destructieve query's zijn. Wat bekend is, is dat er query's zijn met een langere doorlooptijd en dat dit mogelijk kan zorgen voor de vertraging in de database. Maar welke query's dit specifiek zijn is niet bekend.

De resultaten van de query's zullen moeten worden gerealiseerd aan de hand van de oplossing in de voorgaande alinea's. Er zal aan de hand van de resultaten worden bepaald wat op de monitor wordt weergegeven.

5. Proces aanpak

In de onderstaande drie procesfases zijn een aantal methodieken toegepast volgens het HBO-I model. Om het proces zo goed mogelijk te laten verlopen is ervoor gezorgd dat er minimaal één keer per week een overleg was met de bedrijfsbegeleider en één keer in de drie weken met de stagebegeleider. Door dit te doen is het proces goed verlopen [2].

5.1 Analyse

Er is op verschillende manieren onderzocht hoe het probleem kan worden aangepakt. Aan de hand van literatuurstudie is er informatie gezocht over de gestelde deelvragen. Ook is er met medewerkers van Verzuimsignaal gebrainstormd over de deelvragen. Tijdens deze sessies is er gekeken naar wat passende oplossingen of aanpakken kunnen zijn om het gestelde probleem aan te vliegen. Door middel van deze verschillende methodieken konden er duidelijke conclusies worden getrokken en bepaalde keuzes in het onderzoek worden gemaakt.

5.2 Ontwerp

In de ontwerpfase zijn er een aantal requirements en acceptaties voor het systeem opgesteld. Aan de hand van deze requirements zijn een aantal ontwerpen gemaakt, waaronder een schets van het systeem, een component diagram en klassediagrammen. De component diagram en klassediagram zijn op basis van UML gemaakt. Verder is er rekening gehouden met de schaalbaarheid van het systeem. Ook is er met een expert bekeken of alle nodige componenten in de verschillende diagrammen zitten aan de hand van een peer review. Op basis van deze methodieken is er een ontwerp opgesteld dat aansluit bij de wensen van de opdrachtgever.

5.3 Realisatie

Aan de hand van de requirements en de gerealiseerde ontwerpen is er een prototype gemaakt. Dit prototype zal door middel van peer review met een expert worden besproken. Ook is er een system test uitgevoerd om te bepalen of alle eisen van het systeem voldoen aan de gestelde eisen. Ook zal er rekening worden gehouden dat de software schaalbaar moet zijn. Door middel van het ontwerp en het onderzoek is er een passend prototype gerealiseerd. Ook is er gemeten hoeveel tijd het prototype extra kost om uit te voeren.

6. Onderzoek

Bij het afstudeerproject zullen er een aantal zaken moeten worden onderzocht. Aan de hand van de probleemstelling is er duidelijk geworden wat de hoofdvraag is, namelijk: “Hoe kunnen destructieve query’s worden gedetecteerd die worden uitgevoerd op de databases?”. Deze hoofdvraag kan een aantal deelvragen worden opgesplitst namelijk:

- Hoe kunnen de query’s getransporteerd worden voor analyse, zonder hinder op het huidige proces?
- Hoe kunnen de query’s geanalyseerd worden?
- Wat is de definitie van een destructieve query?
- Hoe kunnen de query’s (realtime) worden gemonitord?

6.1 Transporteren query’s

Bij de deelvraag, “Hoe kunnen de query’s getransporteerd worden voor analyse, zonder hinder op het huidige proces?” is er gekeken hoe de query’s kunnen worden getransporteerd van de Verzuimsignaal database/applicatie naar een omgeving buiten de Verzuimsignaal database/applicatie. Bij deze deelvraag zal er rekening moeten worden gehouden met de beperkte opslag. Ook zal er rekening moeten worden gehouden met de grote hoeveelheid query’s die er per seconde worden geproduceerd. Op piekmomenten kunnen dit er meer dan 1 miljoen per seconde zijn. Dit komt doordat er een grote hoeveelheid klanten op hetzelfde moment gebruik maken van het systeem. Er zal bij dit punt hoogstwaarschijnlijk gebruik worden gemaakt van een message broker. Met een message broker kan namelijk een grote hoeveelheid berichten worden afgehandeld, voorbeelden zijn “RabbitMQ” en “Kafka”. Uit het onderzoek zal gaan blijken wat de beste optie zal zijn.

6.1.1 Onderscheppen van query’s

Als eerste stap is er gekeken naar hoe de query’s onderschept kunnen worden. Als dit is onderzocht kan er worden bekeken worden hoe deze data kan worden getransporteerd naar een externe server. Er wordt gekeken naar MaxScale, Logging binnen MariaDB en de DBConnector van de applicaties van Verzuimsignaal.

6.1.1.1 MaxScale

MaxScale is een proxy die zorgt voor een hoge beschikbaarheid. Over de MaxScale lopen ook alle query’s. Het voordeel is dat er niet rechtstreeks in de applicatie aanpassingen hoeven worden gedaan. Een nadeel daarentegen is dat er belangrijke informatie verloren gaat, namelijk de uitvoertijd van een query tot een resultaat, van welke database de data komt en welk cluster er is gebruikt. Aangezien het niet bekend is welke database is kan er niet achteraf een explain ([6.2.1 Explain \(mysql/mariaDB\)\)](#)) worden uitgevoerd op de query die wordt afgevangen.

6.1.1.2 Logging

Binnen mariaDB kan er gebruikt worden gemaakt van de logging. Er kan worden gekozen tussen de “general query log” en de “slow query log” de general query log heeft hetzelfde probleem als de MaxScale. Er kan niet worden bepaald wat de uitvoertijd van een query is. Daarentegen kan er wel worden gezien vanaf welke database de query komt. De slow query log heeft als extra dat de uitvoertijd van de query kan worden gezien en dat de tijd kan worden ingesteld wanneer een query te traag is. Een algemeen nadeel aan de logging is dat de logging voor een heel cluster tegelijk aan moet. Op deze manier wordt de belasting voor een database vrij hoog.

6.1.1.3 DBconnector

De DBConnector is een klasse die binnen de applicatie Verzuimsignaal en DataExchange bevindt. Binnen deze klasse bevindt zich een functie genaamt “executeQuery”. Over deze functie lopen alle query’s van de applicatie. Ook kan er binnen deze functie de tijd van het resultaat van de query worden gemeten. Verder is het mogelijk om de pagina waar de query wordt uitgevoerd op te slaan. Ook kan hier de databasenaam en de clusternaam worden opgehaald.

Aan de hand van al deze gegevens kan er een compleet beeld worden gecreëerd. Doordat het bekend is op welk cluster en database de query wordt uitgevoerd kan er ook achteraf gebruik worden gemaakt van andere query’s bijvoorbeeld, explain ([6.2.1 Explain \(mysql/mariaDB\)](#)).

6.1.1.4 Conclusie

Er is gekozen voor het gebruik maken van de DBConnector. Deze keuze is gemaakt aangezien hier de benodigde gegevens beschikbaar zijn, die nodig zijn om een compleet beeld te creëren. Het nadeel is dat de code van de applicatie zal moeten worden aangepast, maar daarentegen is de data betrouwbaar en compleet.

6.1.2 Message brokers

Er zijn veel verschillende message brokers, waaronder ook veel forks (aanpassingen op een bestaande message broker) van bestaande message brokers. Het doel van een message broker is om een groot aantal berichten tijdelijk te kunnen opslaan om later te verwerken. Vanaf deze queue kunnen de berichten naar verschillende workers worden gestuurd om het proces zo snel mogelijk te laten verlopen. Op deze manier is de omgeving schaalbaar. Er is gekeken naar drie verschillende soorten message queues namelijk RabbitMQ, ZeroMQ en Kafka. In onderstaande hoofdstukken wordt uitgelegd wat de message queues als voor- en nadelen hebben.

6.1.2.1 RabbitMQ

RabbitMQ is een voorbeeld van een message broker. Het principe is vrij simpel. RabbitMQ bestaat uit een publisher, queue en consumer. In onderstaande hoofdstukken zal er uitgelegd worden wat deze componenten inhouden. Ook wordt er gekeken naar verschillende protocollen en de snelheid van RabbitMQ.

6.1.2.1.1 Publisher

Het eerste deel van RabbitMQ is een publisher. Hier kunnen de berichten op prioriteit worden ingedeeld. Wanneer de juiste prioriteit is bepaald zal het bericht naar de juiste queue worden gestuurd. Er kunnen meerdere publishers worden gebruikt mocht dit nodig zijn [4].

6.1.2.1.2 Queue

Wanneer de publisher ervoor heeft gezorgd om de juiste prioriteit aan het bericht te geven komt het bericht in de queue. In de queue wordt het bericht klaargezet voor de consumer. Ook kan er gebruik worden gemaakt van lazy queue. Lazy queue zorgt ervoor dat berichten die niet worden behandeld op de disk worden opgeslagen. Het voordeel hiervan is dat het RAM-geheugen minder wordt belast. Er zal op deze manier meer gebruik worden gemaakt van de disk snelheid. De berichten die worden behandeld worden op het RAM-geheugen verwerkt. Wat ook bekend is van een queue is dat het een single threaded proces is. Er moet dus voor worden gezorgd dat één queue niet te veel input krijgt, anders zal deze queue snel vollopen [3].

6.1.2.1.3 Consumer

Als het bericht vanuit de queue naar de consumer wordt gezonden zal de consumer het bericht afleveren bij een passende worker. Het is mogelijk om meerdere consumers op een queue aan te sluiten. Het voordeel hiervan is dat er meerder consumers dezelfde taak hebben waardoor de queue beter leeg kan worden gehouden. Op deze manier is RabbitMQ ook zeer goed schaalbaar. Wanneer de load hoger wordt kunnen er meer consumers/workers worden bijgeschakeld. Elk bericht is slechts eenmaal beschikbaar. Hierna is het bericht verdwenen. Dit kan voor- en nadelen met zich meebrengen. Het voordeel hiervan is dat het bericht nergens kan blijven zweven en dat er geen overload aan data kan ontstaan. Als nadeel heeft het dat het bericht niet overnieuw kan worden gebruikt. Daarentegen is de vraag of het bericht vaker moet worden gebruikt. In deze case zou het beter zijn om het bericht niet op te slaan. Dus in dit geval is het een voordeel dat de berichten niet worden opgeslagen [4].

6.1.2.1.4 Snelheid

RabbitMQ is zoals eerder aangeven goed schaalbaar. Het is mogelijk om bijvoorbeeld 1 miljoen berichten per seconde te verwerken met ongeveer dertig nodes. Wat het voordeel is dat er weinig nodes kunnen worden gebruikt wanneer het relatief rustig is. Wanneer de load hoger wordt kunnen er meer nodes worden bijgeschakeld [1].

6.1.2.1.5 Protocollen

Bij RabbitMQ kunnen verschillende protocollen worden gekozen. Er zullen een viertal protocollen worden behandeld. AMQP 0-9-1 is standaard geïmplementeerd protocol bij RabbitMQ en wordt dus ook het vaakst gebruikt. Er bestaan ook andere protocollen zoals STOMP, MQTT en HTTP and WebSockets. STOMP is een text-based message protocol en simpel te gebruiken. Een nadeel is dat het vrij lastig is om het op een server te gebruiken. MQTT is een binair protocol. Dit protocol is lichtgewicht en is vooral gemaakt voor het publiceren en inschrijven van berichten en dus niet voor andere soorten berichten. Tot slot is het protocol HTTP beschikbaar. Dit is niet echt een message protocol, maar RabbitMQ kan wel overweg met dit protocol om dit in combinatie met STOMP of MQTT te gebruiken. Alle protocollen op AMQP 0-9-1 zijn als plugin toe te voegen aan RabbitMQ. AMQP 0-9-1 is al standaard geïmplementeerd binnen RabbitMQ en hoeft dus niet als plugin worden meegenomen. Wanneer er wordt gekozen voor RabbitMQ kan wordt er ook gekozen voor het standaardprotocol [4].

6.1.2.2 Kafka

Kafka is een wat complexere message handler. Deze bestaat uit een publisher, queue, consumer en eventueel zookeeper. In onderstaande paragrafen wordt er uitgelegd wat deze onderdelen inhouden.

6.1.2.2.1 Publisher

De publisher van kafka is niet zoals bij RabbitMQ bedoeld om mee te sorteren. Deze zal de berichten aannemen en in het kafka cluster (meerdere kafka nodes inclusief zookeeper) brengen. Binnen in dit cluster zal de data worden gesorteerd in de aangegeven volgorde en ook in de juiste volgorde geconsumeerd [4].

6.1.2.2.2 Consumer

De consumer is daarentegen weer slimmer dan RabbitMQ en heeft ook de mogelijkheid om te sorteren in combinatie met zookeeper. Hierdoor kunnen de berichten die binnenkomen in een vooraf gedefinieerde volgorde worden gezet. Vervolgens zal het bericht worden gezonden naar bijvoorbeeld een microservices/worker die het bericht verder zal afhandelen. Alle berichten die zijn geconsumeerd zijn meerdere malen beschikbaar en zullen niet direct worden verwijderd [4].

6.1.2.2.3 Snelheid

Kafka kan net zoals RabbitMQ een grote hoeveelheid berichten per seconde verwerken. Dit zijn er al snel 1 miljoen per seconden dit kan meer of minder worden. Dit is afhankelijk van de snelheid. Kafka werkt vooral aan de hand van disk I/O. Het voordeel hiervan is dat er in verhouding minder resources nodig zijn. Maar als er hoge snelheden moeten worden gehaald zullen er nieuwe server clusters bij moeten komen om de nodige snelheid te halen [1].

6.1.2.3 ZeroMQ

Verder is er ook gekeken naar ZeroMQ dit is een vrij lichtgewicht systeem en draait op een library. Het principe heeft veel weg van RabbitMQ maar heeft een groot nadeel. Het systeem kan rond de 5 miljoen berichten per seconde versturen, maar kan maar een maximum van zeshonderdduizend berichten per seconde ontvangen. Aangezien de database op piektijden 1 miljoen of meer berichten kan versturen zal dit niet geschikt zijn voor de huidige situatie en is het onderzoek in dit onderwerp ook vrij snel gesloten [6].

6.1.2.4 Conclusie

Als er naar de bovenste drie message queues wordt gekeken is het duidelijk dat er niet voor ZeroMQ wordt gekozen. Dit heeft puur te maken met het aantal berichten die kunnen worden verstuurd. Vervolgens blijft RabbitMQ en Kafka over. Hier zitten voor- en nadelen aan. De keuze is uiteindelijk gevallen op RabbitMQ, aangezien de berichten niet opnieuw hoeven geconsumeerd te worden. Ook is het niet nodig om te sorteren aangezien er echt per bericht wordt gekeken. Er zit dus geen bepaalde volgorde in. Ook is er gekeken naar het huidige applicatielandschap van de applicatie zelf. Hier wordt al gebruik gemaakt van RabbitMQ. Het geniet de voorkeur om vanuit de applicatie zelf RabbitMQ te gebruiken.

6.1.3 Queues

Ook zijn er een aantal queues onderzocht hoe deze functioneren. Voorbeelden van queues zijn: First-Come, First-Served (FCFS), Round Robbin (RR) en Multilevel Feedback Queue. In Het onderzoeksverslag ([11.1 Onderzoeksverslag](#)) zal er dieper worden ingegaan op de verschillende soorten queues die er kunnen worden gebruikt. Dit onderzoek is vooral gebruikt om het principe van een queue te snappen. RabbitMQ regelt het queue mechanisme zelf.

6.2 Query's analyseren

Bij de deelvraag, “Hoe kunnen de query's geanalyseerd worden?” zal er worden gekeken wat opties zijn om de query's te analyseren. Aan de hand van de analyse zal er moeten worden gekeken of de eigenschappen van de query's zo veel mogelijk naar boven worden gehaald. Aan de hand van dit materiaal kan er gebruik worden gemaakt van bijvoorbeeld een statische oplossing of machine learning. In de eerste instantie zal er gebruik worden gemaakt van de statische oplossing en eventueel als er tijd voor is naar een van de andere opties. In het hoofdstuk [6.4 Monitoren](#) zal uitgelegd worden welke statische waarden op gecontroleerd gaat worden.

6.2.1 Explain (mysql/mariaDB)

Om een query te analyseren kan er gebruik worden gemaakt van “Explain”. Explain kan voor een select statement worden gezet. Op deze manier zal er een weergave worden gegeven hoe een query is opgebouwd. Er worden een aantal verschillende velden weergegeven namelijk:

- select_type
- table
- type
- possible_keys
- key_len
- ref
- rows
- extra

In deze kolommen kan er worden gezien wat een query gebruikt en eventueel worden geconstateerd waarom een bepaalde query traag is. Er kan aan de hand van de rows worden berekend hoeveel rijen er worden aangeraakt om tot een antwoord komen. Het beste is dat het aantal rows die worden aangeraakt zo klein mogelijk blijft. Wanneer een query bijvoorbeeld uit vier stappen bestaat met de bijvoorbeeld de volgende rijen aantallen: 83, 1, 1 en 1700. Deze rijen zullen worden vermenigvuldigd en dit geeft aan hoeveel rijen er in totaal worden aangeraakt. In dit voorbeeld is de kans groot dat de eerste stap geen index bevat. Als dit wordt aangepast kan de eerste stap in misschien wel één rij worden gedaan. Het voordeel hiervan is dat er minder rijen worden bekeken/aangeraakt. Waar ook naar kan worden gekeken is de kolom type en extra. Het meest gunstige is dat type op eq_ref staat. Dit houdt in dat het een unieke referentie is. Hierdoor is er geen index nodig aangezien de database direct weet waar die moet zijn. Dit is ook de snelste manier. Ook kan er gekeken worden in de extra kolom. Wanneer het aantal rows hoger is staat er vaak interessante informatie in de extra kolom. Hier kan bijvoorbeeld staan dat er gebruik wordt gemaakt van filesort. Het effect hiervan is dat alle kolommen worden verzameld en vervolgens gesorteerd. Als deze tabel niet al te groot is wordt deze in een tijdelijke tabel opgeslagen. Dit zorgt ervoor dat niet twee keer hetzelfde moet worden opgezocht [5].

6.2.2. Logging

Een andere manier om te analyseren is gebruik maken van de slow query log of de general query log. Dit wordt voor deze case niet gebruikt in verband met de grote hoeveelheden query's en de beperkte opslagcapaciteit. Voor meer informatie over de verschillende loggings kan worden gevonden in het onderzoeksverslag ([11.1 Onderzoeksverslag](#)).

6.3 Definitie destructieve query

Bij de deelvraag “Wat is de definitie van een destructieve query?” is er gekeken naar wat de belangrijkste eigenschappen zijn van een destructieve query. Deze eigenschappen kunnen de analyse weer meer scopen waardoor de analyse specifiek wordt. Mogelijke eigenschappen van een destructieve query kunnen zijn een langere doorlooptijd. In onderstaande tekst zal verder worden uitgelegd wat eigenschappen zijn van destructieve query's.

Als er wordt gekeken naar een query zijn er een aantal belangrijke eigenschappen namelijk: de aantal rijen dat wordt aangeraakt, de tijd dat een query uitgevoerd wordt, het aantal query's die in een actie worden uitgevoerd.

Bij het definiëren van een destructive query is er in overleg met de opdrachtgever gekozen om de uitvoertijd van een query als meetpunt te nemen. Er wordt gekeken naar query's die langer duren dan 5 seconden. Mochten dit geen resultaat opleveren dan zal de tijdschaal naar beneden worden bijgesteld. Het getal wat is gekozen is volledig variabel en zal handmatig worden aangepast aan de hand van de resultaten uit de programmatuur. Op deze manier kunnen de query's met een lange doorlooptijd worden bekeken en beoordeeld waarom deze een lange doorlooptijd hebben. Vaak is er een combinatie in tijd en de aantal rijen die zijn aangeraakt voor het uitvoeren van een query. Ook kunnen er gegevens uit de “explain” worden gehaald. Hier kan bijvoorbeeld worden aangegeven dat er gebruik wordt gemaakt van een temporary table of dat de hele tabel wordt door gezocht. Dit kunnen tekenen zijn van een destructive query.

Met de opdrachtgever is er afgesproken om de tijdsblokken per 0.2 seconde te laten zakken (0.1 seconde en 0.2 seconde). Het tijdvak van 0 seconde zal een enkel vak zijn. Deze keuze is gemaakt aangezien in dit tijdsvak de query's zodanig kort zijn dat dit geen probleem is op het systeem en dus automatisch geen destructive query is op tijd gebaseerd.

Een volgende stap is om de hoeveelheid te bekijken. Deze hoeveelheden worden gebaseerd aan de hand van de duur van een actie. Er zal eerst moeten worden bepaald hoeveel query's er maximaal en minimaal per actie plaatsvinden. Als dit is bepaald kan er een schaalverdeling worden gemaakt wat veel en weinig query's zijn.

Een actie is een gebeurtenis zoals: het inloggen op de Verzuimsignaal app of het zoeken van een werknemer. Deze acties zullen moeten worden gegroepeerd om een duidelijk beeld te krijgen hoeveel query's een actie bevat

Een ander eigenschap waarnaar wordt gekeken is of query's dubbel worden uitgevoerd. Wanneer een query dubbel wordt uitgevoerd is dit vaak niet nodig als dit binnen dezelfde actie gebeurt. Er zal moeten worden gekeken naar de tussenliggende stappen bij het dubbel uitvoeren van een bepaalde stap. Mocht de data tussentijds worden aangepast is de kans groot dat er een bewuste keuze is gemaakt om de query twee of meer keer uit te voeren. Maar als dit niet het geval is, is de kans groot dat de data tijdelijk had kunnen worden opgeslagen binnen de applicatie zelf. Wanneer dit zou worden gedaan kan de belasting op de database worden verminderd.

Op de deelvraag “Wat is de definitie van een destructieve query?” kan het volgende antwoord worden gegeven namelijk, Een destructieve query heeft verschillende eigenschappen namelijk: tijd, de aantal rijen die worden aangeraakt bij het uitvoeren van een query en de aantal query's die worden uitgevoerd binnen een actie. Samen met de opdrachtgever is bepaald dat de tijd de hoofdeigenschap is waarnaar wordt gekeken. Op deze manier zal een destructieve query worden gedefinieerd.

6.4 Monitoren

Bij de laatste deelvraag, “Hoe kunnen de query's (realtime) worden gemonitord?” zal er worden bepaald wat de meest belangrijke waarden zijn en hoe het inzichtelijk kan worden gemaakt wat destructieve query's zijn. Er zal moeten worden bepaald of het mogelijk is om de data realtime te monitoren of dat dit in een soort van rapportage wordt weergegeven eenmaal in een bepaalde tijd. Het belangrijkste is dat er inzicht in de query's wordt getoond welke mogelijk problemen kunnen veroorzaken of kunnen worden geoptimaliseerd om de snelheid van de database te verhogen.

Met de opdrachtgever is er over gesproken wat belangrijk is om weer te geven. Uit het gesprek is naar voren gekomen dat de totalen van waarden een belangrijk meetpunt is. Waar veel waarde aan wordt gehecht is een gehele actie (bijvoorbeeld: inloggen of een koppeling leggen) en hoelang heeft de totale uitvoertijd van een actie geduurd.

Een andere interessante waarde is om te kijken hoe vaak een bepaalde query dubbel wordt uitgevoerd. Deze waarde zal moeten worden gerangschikt van hoog naar laag. Ook zou deze waarde nog kunnen worden opgesplitst in het soort statement (Select, Delete, Update, etc.). Om dit als een waarde te presenteren kan ervoor worden gekozen om het aantal duplicaten van een actie weer te geven. Stel dat een actie uit tien query's bestaat en twee van deze query's bevatten duplicaten. Dan zal dit als twee duplicaten binnen de actie worden weergegeven. Een ander voorbeeld is om het totale tijdverlies te berekenen veroorzaakt door de duplicaten.

Verder is het mogelijk om een lijst weer te geven hoe vaak er gebruik wordt gemaakt van filesort of een temporary tabel. Dit zullen dan uiteraard twee aparte waarden zijn. Aangezien een explain uit meerdere delen kan bestaan is het mogelijk dat beiden soorten hierbinnen zitten. Ook is het mogelijk dat een van deze twee meerdere keren binnen een explain worden gebruikt.

Wanneer de verschillende acties in een grafiek zullen worden getoond door een bepaalde taak te monitoren is het mogelijk patronen te leren kennen in de query's die worden uitgevoerd. Hier kan bijvoorbeeld zijn dat de waarden op dag één lager zijn dan dag twee. Een mogelijke aanleiding is dat de software is aangepast of dat de database veel wordt gebruikt op dat moment. Ook zou het kunnen zijn dat er een query een lange doorlooptijd heeft waardoor bepaalde rijen geblokkeerd worden. Dit is alleen te achterhalen in de slow query log, maar dit is geen optie om te gebruiken.

De vraag "Hoe kunnen de query's (realtime) worden gemonitord?" is dus als volgt beantwoord. Door middel van verschillende totaalwaarden en meerder verschillende eigenschappen kunnen er conclusie worden getrokken uit een bepaalde actie. Door de verschillende uitslagen op te slaan kan er een grafiek worden gemaakt die laat zien wat van patroon in een bepaalde uitvoering zit. Moet deze waarde hoger worden kan er actie worden ondernomen. Ook kan het zijn dat de drukte toeneemt waardoor de waarde hoger is en dit kan worden meegenomen in het patroon.

6.5 Conclusie

De hoofdvraag "Hoe kunnen destructieve query's worden gedetecteerd die worden uitgevoerd op de databases?" is beantwoord in meerdere stappen. De hoofdvraag is in vier delen opgesplitst.

Er is begonnen met het transporteren van de query's die in de applicatie worden uitgevoerd naar een lokale omgeving. Hier is gekozen om dit via de DBConnector van de applicaties van Verzuimsignaal te doen. Deze keuze is hoofdzakelijk gemaakt doordat hier alle benodigde gegevens beschikbaar zijn. Vervolgens zal de afgevangen data moeten worden verplaatst naar de lokale omgeving. Eén belangrijk punt is hier dat de snelheid van de applicatie niet mag verlagen. Hier is gekozen voor RabbitMQ, de belangrijkste reden hier is dat dit veel berichten kan versturen en ontvangen en dat deze makkelijk schaalbaar is. Ook wordt dit al binnen de applicatie gebruikt dus dit geeft een voordeel in de implementatie.

Vervolgens is er verder gekeken naar hoe query geanalyseerd kunnen worden. Als eerste is er gekeken naar de verschillende soorten logging binnen MariaDB maar aangezien dit te belastend is en te veel ruimte kost is er gekeken naar een andere methode. Uiteindelijk is er gekozen voor de explain functie van MariaDB. Hier kan een getransporteerde query verder worden onderzocht en dit geeft een duidelijk beeld.

Ook moest er bepaald worden wat de definitie van een destructieve query is. Er is gekeken naar verschillende eigenschappen, zoals de tijd en het aantal aangeraakte rijen om een query uit te voeren. Er is vooral gefocust op de doorlooptijd van het genereren van een resultaat uit een query. Hier is voor gekozen om dit in tijdvakken in te delen.

Tot slot is er bepaald wat interessant is om te monitoren en hoe dit moet worden weergegeven. Hier is de keuze gevallen om totaalwaarden te genereren, waarbij is te denken aan de totale tijd van een actie en hoeveel duplicaten ervoor komen. Door een test vaker uit te voeren kunnen er patronen worden herkend en kunnen ook piektijden worden gedetecteerd.

Door middel van het toepassen van bovenstaande technieken en onderzoeksresultaten is het mogelijk om destructieve query's te detecteren die op de database plaatsvinden.

7. Ontwerp

7.1 Requirements

Aan de hand van het onderzoek en samen met de opdrachtgever zijn een aantal requirements voor het systeem opgezet. Aan de gestelde requirements zal het systeem worden opgebouwd. Ook zullen de requirements SMART worden gemaakt. Dit houdt in dat de requirement wordt beschreven volgens de volgende criteria, specifiek, meetbaar, acceptabel, realistisch en tijdsgebonden. Er zal ook gebruik gemaakt worden van MoSCoW (Must, Should, Could, Would (not)). Aan de hand van deze methode zal er duidelijk worden gemaakt welke vereisten er aan de software zitten en welke niet.

Requirements		
#	Requirement	MoSCoW
1	Het systeem van Verzuimsignaal mag niet tot nauwelijks hinder/tijdverlies vernemen door de aanpassingen in het systeem.	M
2	Het systeem moet de dataopslag kunnen regelen waardoor er alleen relevante data wordt opgeslagen. Aangezien er op piekmomenten al 1 miljoen query's per seconde kunnen zijn. Dit zou niet haalbaar zijn om op te slaan.	M
3	Het systeem kan de verschillende berichten splitsen in groepen. Dit zal op database, omgeving (ontwikkel, test, acceptatie, productie) en uitvoertijd moeten worden gedaan. De tijd moet als volgt worden ingedeeld, namelijk in de groep 0 seconden, 0.1-0.2 seconde tot en met 4.8-4.9 seconden, dit houdt in, in stappen van 0.2 seconde. Als de tijd boven de 5 seconde komt wordt die in de groep 5+ seconden gedaan.	S

Deze waarden zijn in het onderzoek bepaald. ([6.3 Definitie destructieve query](#))

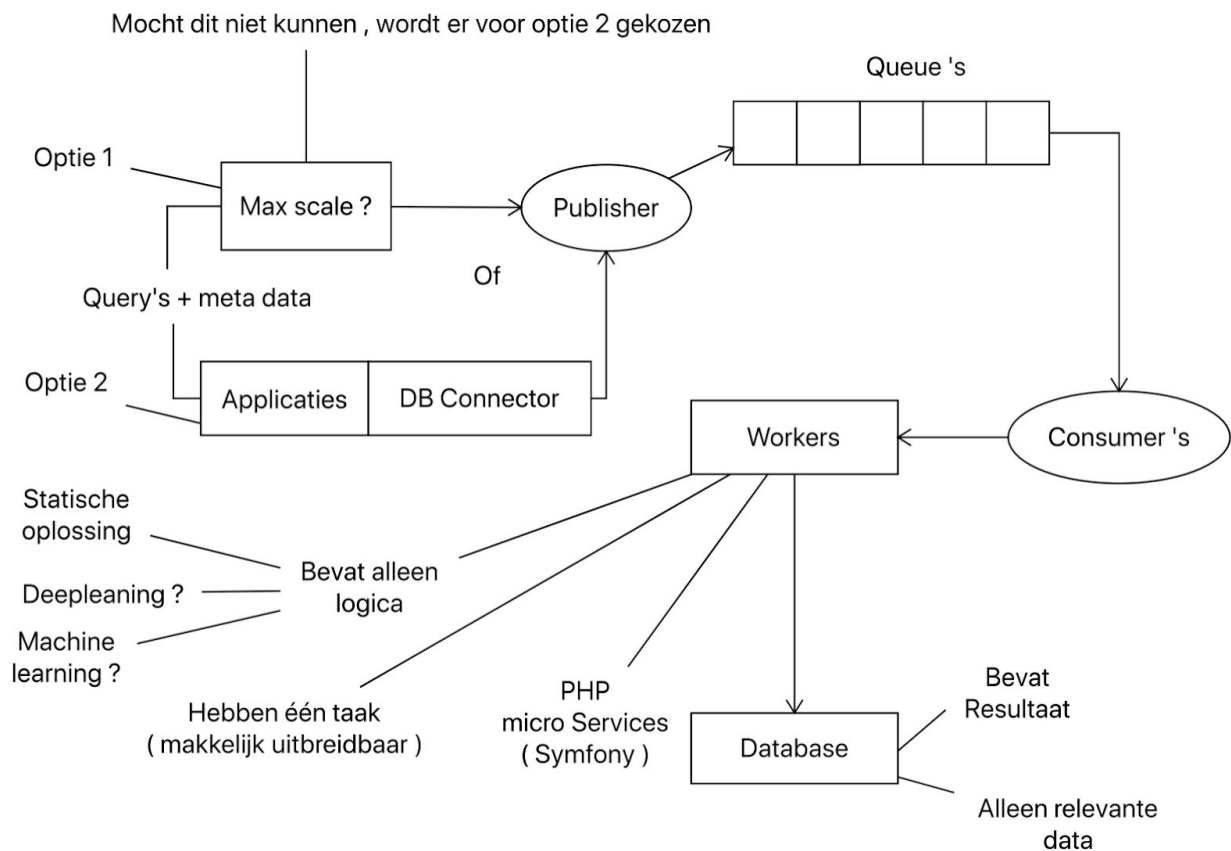
4	Het systeem moet kunnen aangeven op welke pagina/script de query is uitgevoerd. Dit is nodig om te herleiden waar een bepaalde query is uitgevoerd.	Er moet binnen een bericht kunnen worden aangetoond waar een query is uitgevoerd.	S
5	Het systeem moet kunnen aantonen welke query's binnen een actie liggen. Aan de hand hiervan kan er worden gezien hoeveel query's een actie bevat. Op deze manier kan bijvoorbeeld de totale uitvoertijd van een actie worden berekend.	Er moet een groep kunnen worden gemaakt van alle query's die binnen een actie ligt. Dit kan bijvoorbeeld met een unique ID.	M
6	Het systeem moet kunnen aangeven hoeveel rijen er worden bekeken om tot een resultaat te komen.	Wanneer er een getal wordt weergegeven met het totaal aantal rijen dat wordt aangeraakt.	S
7	De applicatie wordt geschreven in PHP	Dit is een verplichting aangezien het bedrijf standaard hier mee werkt	M
8	Het systeem moet voldoende data moeten kunnen verplaatsen. Het mag niet het geval zijn dat er een opstopping aan data ontstaat.	Wanneer er een grote hoeveelheid aan data wordt verstuurd mag dit niet voor vertraging zorgen.	M

7.2 Schets applicatie

In afbeelding 1 is een schets weergegeven hoe de opstelling eruit komt te zien om query's te detecteren. Er zal worden begonnen om de testopstelling via optie 2 te creëren (zoals in het onderzoek is aangegeven [6.1.1 Onderscheppen van query's](#)). Optie 2 houdt in dat er rechtstreeks in de Verzuimsignaal applicatie een aanpassing wordt gedaan in de DBConnector klasse.

Vervolgens zullen de resultaten van de DBConnector aan de publisher worden aangeboden en zal de data in een queue worden gezet. Wanneer de queue wordt aangeroepen door de consumer zal de queue leeglopen en zal de consumer de data aan de worker aanbieden om het te verwerken.

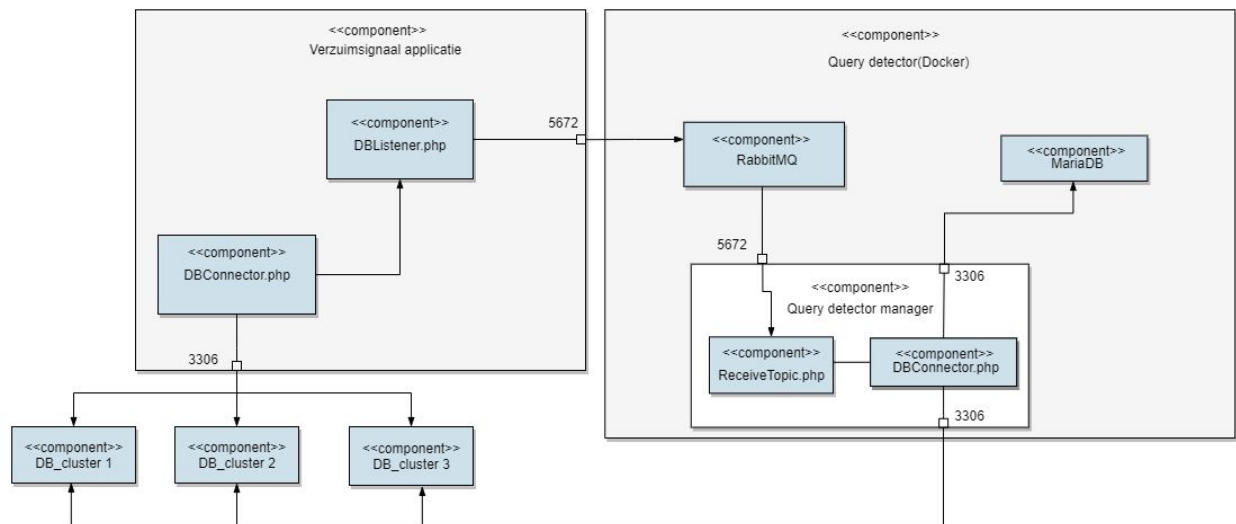
De aangeboden relevante data zal naar een database worden gestuurd en worden opgeslagen voor verdere analyse. Deze data zal via een statische oplossing worden geanalyseerd. Bijvoorbeeld de totale tijd of het aantal rijen die worden aangeraakt. Mocht er genoeg tijd over zijn wordt er eventueel nog ingezoomd op machine- en/of deep learning.



Afbeelding 1: schets applicatie

7.3 Query detector

Met onderstaande prototype kunnen de query's van de Verzuimsignaal applicatie worden getransporteerd naar de query detector. Ook wordt ervoor gezorgd dat de messages worden verwerkt en dat deze in de lokale database terugkomen voor analyse. Er kan via een route worden gefilterd op de databasenaam, environment en de uitvoertijd van een query (requirement 3, [7.1 Requirements](#)). Dit wordt ook verder beschreven in "[8.1.3 Exchange declare of queue declare](#)".



Afbeelding 2: component diagram

7.3.1 DBConnector

De DBconnector van Verzuimsignaal is een klasse die ervoor zorgt dat de toegestuurde query's een resultaat krijgt. Deze klasse heeft namelijk een verbinding met de desbetreffende database. Aangezien over deze klasse alle query's gaan is ervoor gekozen om via deze klasse de verschillende query's te transporteren naar RabbitMQ binnen de "Query detector". De keuze om het naar de query detector te sturen heeft te maken met de belasting op de Verzuimsignaal applicatie, zoals opgenomen in requirement 1 ([7.1 Requirements](#)).

Ook de Query detector manager bevat een DBConnector. Deze DBConnector kan verbinding maken met de lokale MariaDB maar ook de database van Verzuimsignaal. Dit heeft te maken met het proces in de ReceiveTopic klasse. Hier zal in kop [7.3.3 ReceiveTopic](#) verder op in worden gegaan waarom deze keuze is gemaakt.

7.3.2 DBListener

De DBListener klasse binnen de verzuimsignaal applicatie is een verlengstuk aan de DBConnector klasse. De klasse DBListener zorgt ervoor dat de data wordt gebundeld als een message en dat deze vervolgens kunnen worden verstuurd naar RabbitMQ. Op deze manier zullen de query's worden getransporteerd naar de "Query detector". De keuze om

de messages als een bundel te maken heeft te maken met requirement 1 ([7.1 Requirements](#)). Door middel van een bundel wordt de channel van RabbitMQ minder vaak gebruikt. Dit scheelt in de belasting op de Verzuimsignaal applicatie.

7.3.3 ReceiveTopic

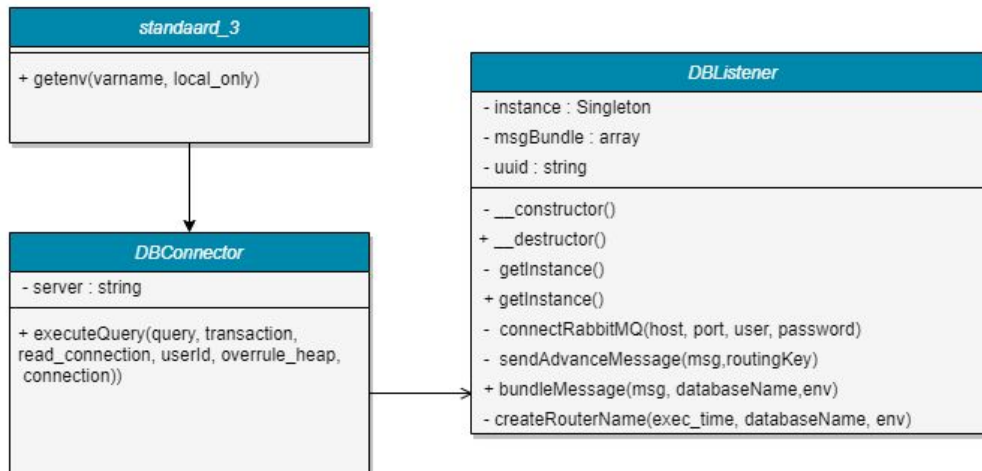
De ReceiveTopic is de consumer van RabbitMQ. Deze klasse zal ervoor zorgen dat de toegestuurde berichten worden ontvangen en verwerkt worden. Zoals in de kop “[7.3.1 DBConnector](#)” is genoemd ligt er een verbinding tussen de externe omgeving en de lokale omgeving. Deze keuze is gemaakt om de “Explain queries” binnen de Query detector te draaien. Het grote voordeel hieraan is, dat de Verzuimsignaal applicatie niet wordt belast door een extra query. Dit is dus ook direct de reden waarom de locale omgeving bij de database van Verzuimsignaal moet kunnen komen.

7.3.4 MariaDB - Query detector

Zoals aangegeven in het component diagram (afbeelding 2) zal er ook gebruik worden gemaakt van MariaDB. Deze database zal alle data verzamelen die door de ReceiveTopic wordt ontvangen. Door middel van deze database kan de data verder worden geanalyseerd zonder dat deze verloren gaat. De database bevat de eigenschappen die zijn aangegeven in de definitie van een destructieve query ([6.3 Definitie destructieve query](#)). Er zullen voor de database verschillende soorten query's worden gecreëerd die er zullen zorgen dat er een beter inzicht komt in de data die wordt toegevoegd. In hoofdstuk [7.6 Output query resultaten](#) zal hier dieper op in worden gegaan.

7.4 Aanpassingen Verzuimsignaal applicatie

In onderstaande klassediagram is een klein deel van de Verzuimsignaal applicatie te zien. Er is vooral ingezoomd op de toegevoegde klasse DBListener, een klein deel van de DBConnector en standaard_3. De standaard_3 klasse wordt gebruikt door de functie executeQuery, standaard_3 zal mee worden genomen bij de DBconnector ([7.4.1 DBConnector](#))



Afbeelding 3: klassediagram - aanpassingen Verzuimsignaal applicatie

7.4.1 DBConnector

De klasse DBconnector bevat meerdere functies. Er is voor gekozen om binnen één functie te werken, namelijk executeQuery. Over deze functie worden alle query's uitgevoerd. Dit is ook direct de reden waarom hier een paar aanpassingen in zijn gedaan.

Er is voor gezorgd dat de tijd van het uitvoeren van een query kan worden gemeten binnen de functie en de starttijd van de query. Ook kan er vanaf hier achterhaald worden op welke omgeving de query's worden gedraaid (ontwikkel, test, acceptatie of productie). Deze informatie wordt via de klasse standaard_3 opgehaald. Verder is er gebruik gemaakt van globale variabelen, zoals: de server naam (cluster) en script naam (de pagina van uitvoeren). Deze gegevens zijn nodig om later een destructive query te definiëren.

Deze data wordt doorgestuurd naar klasse DBListener. Binnen deze klasse is een functie genaamd bundleMessage. Binnen deze functie zal de data verder worden verwerkt. In het volgende hoofdstuk "[7.4.2 DBListener](#)" zal hier dieper op in worden gegaan.

7.4.2 DBListener

De klasse DBListener bestaat uit een aantal verschillende functies, namelijk: `__construct`, `__destruct`, `getInstance`, `connectRabbitMQ`, `sendAdvanceMessage`, `bundleMessage` en `createRouterName`.

Bij het ontwerpen van de klassediagram is er rekening gehouden met de verschillende Requirements ([7.1 Requirements](#)). Als eerste is er gezorgd dat er zo min mogelijk belasting/tijdsverlies op de DBConnector is van de Verzuimsignaal applicatie. Dit is gedaan met behulp van de functie bundleMessage. Deze klasse zorgt ervoor dat de berichten in een cluster van 100 berichten worden verstuurd waardoor de RabbitMQ channel minder vaak hoeft worden te gebruikt. Het aantal berichten is variabel en kan worden aangepast mocht dit nodig zijn.

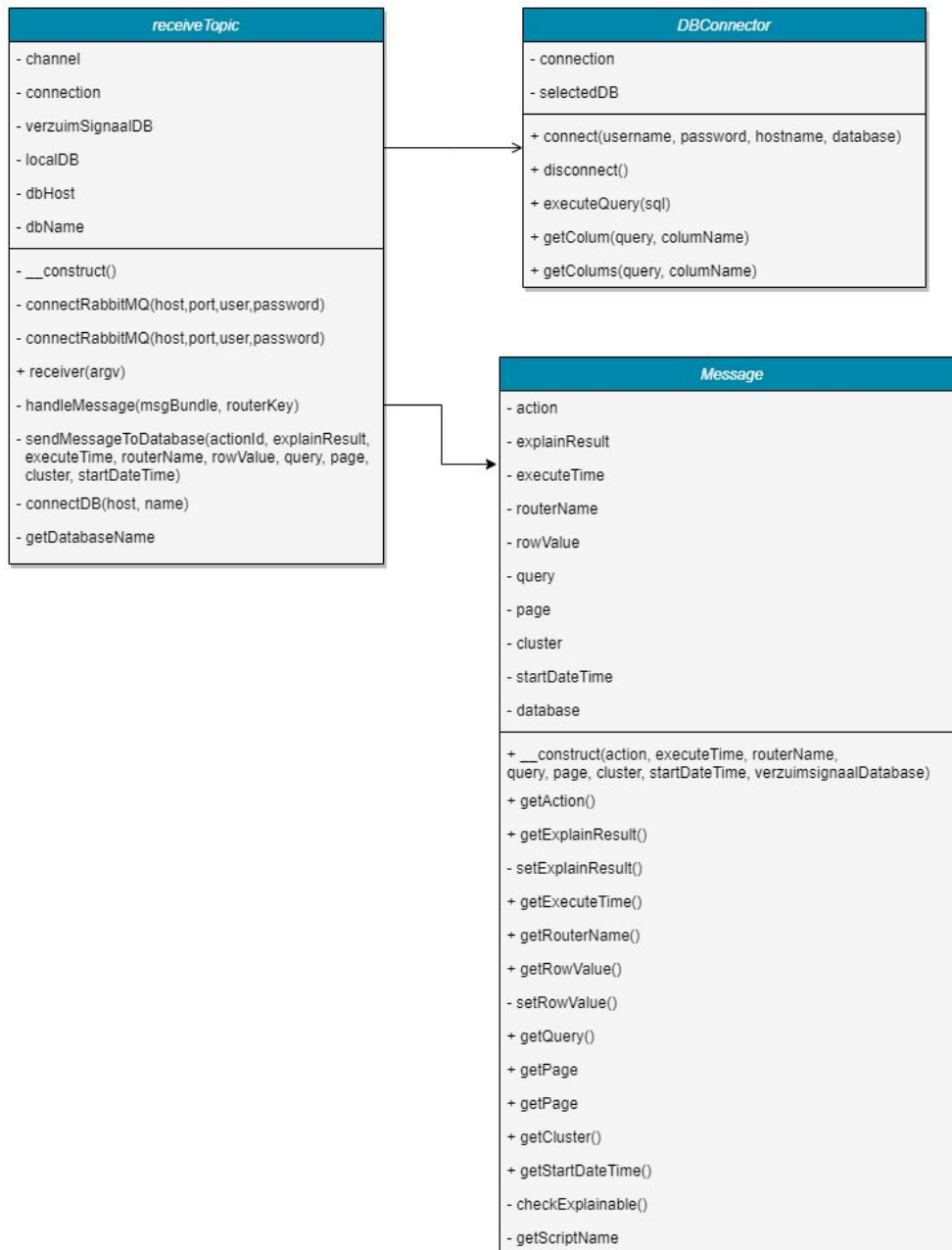
Ook is er een klasse gerealiseerd voor het maken van een router naam. Deze is nodig om de berichten te categoriseren in verschillende groepen. Zoals in requirement 3 ([7.1 Requirements](#)) is aangegeven. De router is als volgt opgebouwd, databasenaam, omgeving en uitvoertijd.

In de constructor wordt ervoor gezorgd dat er een unique id wordt gecreëerd voor een actie. Dit id zal blijven bestaan totdat de DBListener klasse wordt gedestruct. Dit is volgens requirement 5 ([7.1 Requirements](#)).

Voor een meer gedetailleerde uitleg over de verschillende klassen en functies kan het technische ontwerp worden geraadpleegd ([11.2 Technisch Ontwerp](#)).

7.5 Query detector manager

In het onderstaande klassediagram is een weergave gegeven van de applicatie Query detector manager. Deze applicatie heeft een aantal taken, namelijk: Het filteren van berichten, het compleet maken van een bericht, het consumeren van berichten en het versturen van de data naar de database.



Afbeelding 4: klassediagram - query detector manager

7.5.1 DBConnector

De DBConnector heeft de mogelijkheid om bij de Database van Verzuimsignaal te komen maar ook bij de lokale database. De DBConnector moet bij de database van Verzuimsignaal komen om over de binnengekomen query's een explain te doen. De keuze om dit pas hier te doen heeft te maken met Requirement 1 ([7.1 Requirements](#)). Doordat hier pas de query wordt uitgevoerd wordt de Verzuimsignaal applicatie niet belast voor het uitvoeren van de query. De DBConnector is gemaakt om query's uit te voeren en eventueel waarnodig antwoord terug te geven.

Voor meer informatie over de DBConnector kan er worden gekeken in het technisch ontwerp ([11.2 Technisch Ontwerp](#)).

7.5.2 ReceiveTopic

De klasse ReceiveTopic heeft de mogelijkheid om berichten te consumeren van de RabbitMQ server die gestuurd worden door de DBListener in de Verzuimsignaal applicatie. Het is mogelijk om de ReceiveTopic klasse meerdere keren te starten. Het voordeel hiervan is dat er door meerder applicatie de berichten kunnen worden opgehaald. Op deze manier is het gemakkelijk opschaalbaar.

In de functie `receivemessage` worden alle functie die in deze klasse aangeroepen. De `receive` klasse ontvangt het bericht van RabbitMQ en deze zal vervolgens worden verwerkt in de Message klasse.

Ook is het mogelijk om aan te geven aan de hand van de route wat wel of niet naar de database zoals aangegeven in requirement 2 ([7.1 Requirements](#)).

Voor meer informatie over dit klassediagram kan er worden gekeken in het technisch ontwerp ([11.2 Technisch Ontwerp](#)).

7.4.3 Message

De message klasse zal dienen om een message compleet te maken en vervolgens te versturen naar de database. Een bericht bestaat uit de volgende eigenschappen namelijk:

- het actie id, dit geeft aan of een bericht binnen een actie valt;
- explain resultaat, deze zal worden gevuld met de de uitkomst van de explain query als dit mogelijk is;
- de uitvoertijd van een query
- de routenaam zoals aangeven in requirement 3 ([7.1 Requirements](#));
- de waarde van het aantal aangeraakte rijen bij het uitvoeren van een query;
- de uitgevoerde query;
- de pagina waar de query is uitgevoerd;
- de database cluster die is gebruikt;
- de datum en tijd wanneer de query is uitgevoerd.

Er zal binnen deze klasse worden gekeken of er over de query een explain kan worden uitgevoerd. Als dit het geval is zal er uit het explain resultaat de aantal aangeraakt rijen worden bepaald zoals aangegeven in requirement 6 ([7.1 Requirements](#)).

Met deze gegevens is er genoeg informatie om te bepalen of query's destructieve eigenschappen vertonen.

7.6 Output query resultaten

In onderstaande tabellen wordt een visuele weergave weergegeven welke query's er worden gecreeërd en welk resultaat er wordt verwacht.

7.6.1 Aantal query's per actie

Hieronder is een voorbeeld hoe het aantal acties kan worden weergegeven. Op deze manier kan er worden gezien of er veel of weinig query's binnen een actie worden uitgevoerd.

action_id	aantal query's per actie
id van de actie	Waarde

Tabel 1: Aantal query's per actie

7.6.2 Aantal duplicaten

Hierdoor worden de queries inzichtelijk die dubbel worden uitgevoerd. Het aantal duplicaten zal worden gesorteerd worden van hoog naar laag. De duplicaten zullen in een paar categorieën worden opgesplitst namelijk: select, delete, update, insert, create en overig. Ook zullen alleen duplicaten worden opgeteld worden als deze binnen dezelfde actie liggen. Op deze manier kan er worden geconstateerd dat query's onnodig vaak op de database worden uitgevoerd. Dit zou eventueel voor vertraging kunnen zorgen.

action_id	query	aantal duplicaten
id van de actie	De query	De queries die meerdere keren wordt uitgevoerd binnen een actie.

Tabel 2: Aantal duplicaten

7.6.3 Doorlooptijd van een query

Op deze manier is het inzichtelijk wat de doorlooptijd is van een query binnen een actie. Er zal van lange doorlooptijd na de kortste doorlooptijd worden gefilterd. Op deze manier kan er snel worden gezien welke query's een langere doorlooptijd hebben. Ook zullen de row_value en de explain_json worden weergegeven. Dit zijn eigenschappen die eventueel oorzaak zijn van de langere doorlooptijd.

action_id	row_value	explain_json	execute_time (doorlooptijd)
id van de actie	De aantal rijen die worden gebruikt om tot een resultaat te komen	De explain output	Waarde

Tabel 3: Doorlooptijd van een query

7.6.4 Rij waarde van een query

In tegenstelling tot de vorige wordt hier gekeken van hoog naar laag wat de rijwaarde is. Hier zal ook de doorlooptijd en explain_json worden meegenomen. Vanaf dit punt kan er beoordeeld worden of deze twee waarden een relatie met elkaar hebben. Ook zou het kunnen zijn dat de doorlooptijd nu nog niet lang is maar dat er preventief maatregelen kunnen worden genomen omdat bijvoorbeeld een index niet is gezet in een databasetabel. Dit zou een mogelijke oorzaak kunnen zijn van een hoge row_value.

action_id	execute_time (doorlooptijd)	explain_json	row_value
id van de actie	Waarde	De explain output	De aantal rijen die worden gebruikt om tot een resultaat te komen

Tabel 4: rij waarde van een query

7.6.5 Explain bevat filesort of temporary tables

Als eerste zal er worden gezorgd dat er binnen een action_id gekeken. Vervolgens zal de data worden gesorteerd worden op de rij waarde. Daarna zal er worden gefilterd op filesort of temporary tables. Vaak is het zo dat deze waarden een verband hebben. Wanneer er gebruik wordt gemaakt van filesort heeft vaak de tabel geen index. Dit kan resulteren in een hogere rij waarde. Dit is ook de reden waarom hierop wordt ingezoomd.

action_id	execute_time (doorlooptijd)	explain_json	row_value
id van de actie	Waarde	De explain output	De aantal rijen die worden gebruikt om tot een resultaat te komen

Tabel 5: explain bevat filesort of temporary tables

7.6.6 Conclusie

Aan de hand van de bovengenoemde query's kan er inzichtelijk worden gemaakt wanneer een query een afwijkend gedrag vertoont. Een voorbeeld is dat een query die een bepaald id verwijderd honderd keer wordt uitgevoerd. Dan is de vraag waarom dit wordt gedaan en of dit wel nodig is. Ook kan het zijn dat een query een lange doorlooptijd is. Dan kan er worden gekeken naar twee punten. Geeft de explain aan dat er bijvoorbeeld een index niet is geplaatst. Ook kan het zijn dat het druk is op dat moment op de database.

7.6.7 Graylog

Om de queries grafisch te visualiseren kan er gebruik worden gemaakt van graylog (dit is een logging systeem dat binnen verzuimsignaal B.V. wordt gebruikt). Met deze applicatie kunnen grafieken, staafdiagrammen en andere soorten worden weergegeven. Dit is nog niet gerealiseerd in het prototype. Op het moment zullen de getallen worden weergegeven in een tabel binnen MariaDB. Het zal wel een goede vervolgstap zijn op het huidige prototype. Met deze uitbreiding kan er sneller worden gezien wat er gaande is binnen de databases.

8. Realisatie

Tijdens de realisatie is het prototype genaamd “Query detector” gecreeërd. Deze is gebaseerd op de keuzes die in het ontwerp zijn gedaan ([7. Ontwerp](#)). Ook is ervoor gezorgd dat er bij de verschillende requirements testen zijn uitgevoerd. Verder is er waar mogelijk gebruik gemaakt van unit testen.

8.1 verzuimsignaal applicatie

8.1.1 Opbouw bericht

In de message die wordt verstuurd naar RabbitMQ zitten de volgende gegevens:

- query
- uniek id per actie reeks
- tijd van uitvoeren van een query
- de databasenaam (verwerkt in de topic naam)
- het cluster van de database
- de starttijd van het uitvoeren van de query
- de pagina
- de environment waar het wordt op uitgevoerd (verwerkt in de topic naam)

Het belangrijkste wat mee wordt gestuurd is de query die op de applicatie wordt uitgevoerd. Alle soorten query's worden over de lijn verstuurd, het ligt alleen aan de filter welke berichten er worden gebruikt of niet. Verder is ook de doorlooptijd een belangrijk gegeven. Hier meer kan worden gezegd hoeveel tijd een query in beslag neemt

Ook wordt er per actie een uniek id meegegeven. Een actie houdt in dat het proces inloggen onder 1 ID valt. Met deze manier kan er inzichtelijk worden gemaakt hoeveel query's er worden gebruikt bij een actie.

De tijd van een query wordt bepaald aan de hand van de tijd dat een query er over doet om een resultaat weer te geven. De tijd zal ook worden gebruikt bij de topic van RabbitMQ (hier zal later dieper op in worden gegaan ([8.1.3 Exchange declare of queue declare](#))). Deze tijd zal een duidelijk inzicht geven hoe snel een query is.

Verder wordt ook de databasenaam opgehaald, deze wordt niet aan de message toegevoegd maar aan de topic, op deze manier kan er worden gefilterd op de databasenaam, de tijd wordt daarentegen wel in de message meegestuurd aangezien deze in groepen in de topic wordt gezet en de exacte tijd in de message. Ook wordt in de topic de environment meegegeven. Dit kan bijvoorbeeld “dev” zijn.

Tot slot wordt de clusternaam van de database in de message meegegeven. Deze is in combinatie met de databasenaam het pad naar de juiste database. Deze gegevens zijn

nodig om een explain achteraf op de database uit te voeren om de laatste informatie te verzamelen.

Met deze gegevens is er een duidelijk beeld van de query in combinatie met de rondom liggende gegevens hiervan. Met deze gegevens zal er worden geanalyseerd wat leidt tot destructive query's.

In de applicatie Verzuimsignaal bevindt zich een klasse "DBConnecotor". Deze klasse gaat gebruikt worden voor het transporteren van de query's. Ook is er een nieuwe klasse gegenereerd genaamt "DBListner", hier wordt ervoor gezorgd voor een stream connectie naar RabbitMq en zullen de gegevens worden verzonden naar de Queue.

8.1.2 Message bundels

Er is gekozen om gebruik te maken van een bundel van messages. Dit heeft als voordeel dat er minder berichten naar de queue wordt gestuurd en dat de snelheid in de applicatie maximaal blijft. Elke keer als er een bericht wordt gepubliceerd over een channel kost dit een klein beetje tijd. Wanneer er bijvoorbeeld 700 berichten door een applicatie worden verstuurd in ongeveer 5 seconde kan dit voor vertraging zorgen. Aan de hand van een message bundel zal de channel minder vaak worden gebruikt en dit heeft minder effect op de snelheid. Ook is deze keuze belangrijk op het verdere verloop gebaseerd. Als er bijvoorbeeld tienduizend clients worden gebruikt zullen de berichten ook zevenhonderd maal tienduizend worden uitgevoerd dit zal resulteren in ongeveer zevenhonderdduizend query's. Door het gebruik van de bundel zal de channel ongeveer honderd keer minder gebruikt worden. De bundel grootte is aanpasbaar naar voorkeur. Zo kan ervoor een kleinere deel factor worden gekozen waardoor er meerdere kleinere bundels over de lijn worden gestuurd.

8.1.3 Exchange declare of queue declare

Er zijn meerdere manieren om berichten te sturen. Er kan worden gekozen voor het sturen naar een specifieke queue. Op deze manier worden alle berichten naar de queue gestuurd ongeacht of ze wel of niet worden geconsumeerd. Het nadeel hieraan is dat er kans is dat een queue te veel data heeft en dat er een probleem ontstaat aan ruimte. Een tweede optie is om gebruik te maken van een Exchange, Deze kan op twee manieren worden gebruikt namelijk Direct of Topic based. Als er gekozen wordt voor "direct exchange" kan er maar één soort worden meegegeven (bijvoorbeeld: hoog, midden, laag). Wanneer er gebruik wordt gemaakt van topic exchange kunnen er meerdere soorten worden meegegeven (bijvoorbeeld: database1.laag, database1.midden, database2.laag).

In de huidige software is er gekozen voor topic exchange. De keuze is gemaakt op basis van de manier van meegeven van data. Het eerste stuk zal de databasenaam bevatten, het tweede deel de environment en het derde deel de snelheid van de query. Op deze manier kunnen bijvoorbeeld de snelle queues buiten beschouwing worden gehouden. Een ander voordeel is dat als er geen consumer is dat de data direct wordt weggegooid en dus niet in een queue komt. Dit zou ook direct een nadeel kunnen zijn aangezien de data niet allemaal beschikbaar is voor een latere controle. Alleen is dit nadeel niet van toepassing in

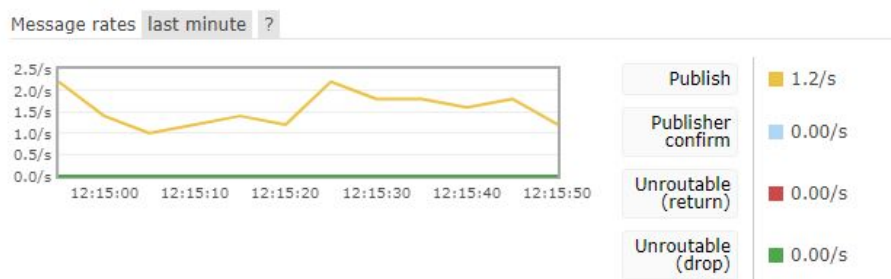
de huidige opdracht aangezien er juist moet worden opgepast met de hoeveelheid data die wordt opgeslagen.

Een ander voordeel aan de verschillende topics is dat deze naar verschillende queues kunnen worden gestuurd. Aangezien een queue single threaded is, is het op deze manier makkelijker om de load te balanceren ([6.1.2.1.2 Queue](#)).

8.2 RabbitMQ

Binnen de Query detector is er een RabbitMQ server opgezet. Dit is met behulp van docker gedaan. Hier kan via een docker-compose file de omgeving worden opgezet. Er wordt gebruikt gemaakt van de TCP-poort die bevindt zich op poort 5672 zoals aangegeven in het ontwerp ([7.3 Query detector](#)). Ook kan er via de grafische omgeving worden gezien hoeveel berichten er op de queue komen per seconde.

In onderstaande afbeelding wordt weergegeven dat alle berichten worden gepubliceerd. Aangezien er is voor gekozen dat alle berichten worden geconsumeerd zal er niets in de Unrouteable (drop) worden weergegeven. Als daar berichten komen betekent dat er op dat moment geen route voor beschikbaar is en dat de berichten worden verwijderd.



Afbeelding 5: RabbitMQ grafiek - voorbeeld van belasting

8.3 Query detector manager

De Query detector manager dient als een koppeling tussen RabbitMQ en MariaDB. Deze applicatie zal ervoor zorgen dat de berichten worden geconsumeerd. Wanneer het bericht is geconsumeerd zal die de ontbrekende data toevoegen. Vervolgens zal het compleet gemaakte bericht naar MariaDB worden gestuurd om daar verdere analyse te doen op de query's.

8.3.1 Consumeren berichten

In de query detector zal de gezonden message worden geconsumeerd. Bij het consumeren moet er worden aangegeven naar welk topic wordt geluisterd. Er bestaan een aantal wildcards namelijk: de asterisk "*" en hashtag "#". De asterisk "*" kan dienen om een deel van de route een wildcard te geven. Een voorbeeld is "database*.0.1-0.2". Hier wordt aangegeven dat het niet uitmaakt welke environment er wordt gekozen. Een voorbeeld met de hashtag kan zijn "#" of "#.0.1-0.2". Bij het eerste voorbeeld mogen alle topics naar de

queue. Bij het tweede voorbeeld maakt de database en de environment variabele niet uit en wordt er alleen naar een bepaald tijdsvak gekeken.

8.1.2 Worker

Wanneer het bericht is geconsumeerd zal het worden doorgestuurd naar een worker. De worker zorgt ervoor dat de inkomende message verder wordt afgehandeld. Als eerste zal er van de query de explain gegevens worden opgehaald worden mits dit een select, delete of update is. Het is namelijk niet mogelijk om op andere soorten query's een explain uit te voeren. Mocht dit niet het geval zijn wordt deze stap over geslagen. In de volgende stap zullen de gegevens door worden gezet naar een database met de gegevens die zijn binnengekomen. Zie hoofdstuk "[8.1.1 Opbouw bericht](#)" voor de gegevens die in de database komen. Als extra veld is het explain resultaat in JSON toegevoegd. Door middel van deze gegevens kan er een duidelijk beeld worden weergegeven wat bijzondere query's zijn en wat van aparte eigenschappen deze hebben. Een compleet bericht ziet er dus ook als volgt uit:

- actie id
- explain gegevens in json formaat
- doorlooptijd van een query
- rij waarde
- routernaam
- query
- pagina
- cluster naam
- start tijd van een query

8.4 MariaDB

Binnen MariaDB wordt de data opgeslagen binnen een tabel. Deze tabel bevat de gegevens die zijn gecreëerd binnen een bericht en worker. In hoofdstuk [8.1.2 Worker](#) worden alle gegevens van een bericht weergegeven. Op deze gegevens zijn verschillende query's uitgevoerd zoals aangegeven in het ontwerp hoofdstuk [7.3.4 MariaDB - Query detector](#).

In onderstaande paragrafen zullen weergaven van de uitgevoerde queries worden weergegeven met de resultaten.

8.4.1 Aantal query's per actie

In onderstaande actie is een weergave van de aantal query's van een actie. Op het moment bevat de eerste actie over de honderdduizend query's de vraag die hier naar boven komt is of het aantal ook minder kan worden. Is er een mogelijkheid dat een query onnodig vaak wordt gebruikt. Er zal daarom moeten worden gekeken naar de aantal duplicaten binnen een actie. Zier hiervoor [8.4.2 Aantal duplicaten](#).

action_id	aantalActies
5ee202b1e09b25.01034714	115305
5ee207c43c33c9.06681085	74134
5ee1fd529b7d88.49170165	49086
5ee2078a46e290.64390251	31320

Afbeelding 6: aantal query's per actie

8.4.2 Aantal duplicaten

In het volgende resultaat is inzichtelijk geworden dat er query's die rechtstreeks op een id zoeken meerdere keren worden uitgevoerd. De kans dat de waarde aanpast binnen een actie is mogelijk. Daarentegen, wanneer een actie meer dan tweeduizend keer wordt uitgevoerd verdacht. Als de query minder vaak wordt aangeroepen zal de database minder worden belast.

action_id	query_output	duplicates
5ee202b1e09b25.01034714	SELECT id. moederbedrijf id FROM bedrijf WHERE id = '75156' LIMIT 1	2893
5ee202b1e09b25.01034714	SELECT id. moederbedrijf id FROM bedrijf WHERE id = '2379' LIMIT 1	2815
5ee202b1e09b25.01034714	SELECT id. moederbedrijf id FROM bedrijf WHERE id = '1162' LIMIT 1	2790
5ee202b1e09b25.01034714	SELECT id. moederbedrijf id FROM bedrijf WHERE id = '193' LIMIT 1	2683
5ee202b1e09b25.01034714	SELECT id. moederbedrijf id FROM bedrijf WHERE id = '136370' LIMIT 1	2579

Afbeelding 7: aantal duplicaten

8.4.3 doorlooptijd van een query

Ook is er gekeken naar de doorlooptijd (execute_time) van een query. Er is ook wederom op een enkele actie ingezoomd. De eerste query heeft de langste doorlooptijd maar daarentegen is de rij waarde relatief laag. Aangezien het een Delete query is het mogelijk dat er veel data wordt verwijderd wat een wat langere tijd kan duren. Wat opvalt aan de tweede query is dat de rij waarde relatief hoog is. Een oorzaak kan zijn dat er gebruik wordt gemaakt van een temporary tabel of filesort. In hoofdstuk [8.4.5 Explain bevat filesort of temporary tables](#) zal hierop teruggekeken worden.

action_id	row_value	explain_json	query_output	execute_time
5ee202b1e09b25.01034714	3	{\"table\":\"t\",\"kev\":\"verzuimmelding id\"...	DELETE t.d FROM taken...	0.073107957839966
5ee202b1e09b25.01034714	83580	{\"table\":\"partner protocol\",\"kev\":\"PRI...	SELECT protocoltaak.id AS origineel protocol...	0.063425064086914
5ee202b1e09b25.01034714	0	null	INSERT IGNORE INTO 'content arrav 3458' S...	0.059505939483643
5ee202b1e09b25.01034714	0	null	TRUNCATE TABLE 'content arrav 3458';	0.057484149932861
5ee202b1e09b25.01034714	0	null	TRUNCATE TABLE 'status 3458';	0.054949045181274

Afbeelding 8: Doorlooptijd van query

8.4.4 Rij waarde van een query

Verder is er gekeken of de rij waarde en tijd wat met elkaar te maken hebben. Daarom is ook de rij waarde bekeken en gesorteerd van hoog naar laag. Wat hier direct opvalt is dat de eerste query dezelfde query is die als tweede staat bij hoofdstuk [8.4.3 doorlooptijd van een query](#). Hierdoor kan er geconcludeerd worden dat een hoog aantal rijen een verband heeft met de doorlooptijd.

action_id	execute_time	explain_json	query_output	row_value
5ee202b1e09b25.01034714	0.063425064086914	{\"table\":\"partner protocol\",\"kev\":\"PRI...	SELECT protocoltaak.id AS or...	83580
5ee202b1e09b25.01034714	0.0076291561126709	{\"table\":\"content arrav 3458\",\"kev\":\"...	SELECT id FROM 'content arrav...	4492
5ee202b1e09b25.01034714	0.0042932033538818	{\"table\":\"loa 3458\",\"kev\":\"null\",\"kev le...	SELECT loa FROM 'loa 3458'	4327
5ee202b1e09b25.01034714	0.0045261383056641	{\"table\":\"loa 3458\",\"kev\":\"null\",\"kev le...	SELECT loa FROM 'loa 3458'	4327
5ee202b1e09b25.01034714	0.0045700073242188	{\"table\":\"loa 3458\",\"kev\":\"null\",\"kev le...	SELECT loa FROM 'loa 3458'	4327

8.4.5 Explain bevat filesort of temporary tables

Tot slot is er gekeken of de explain gebruikt maakt van een temporary tabel of filesort. Wanneer dit het geval is kan dit invloed op de aantal gebruikte rijen en de doorlooptijd van de query. Ook hier komt dezelfde query naar boven die een wat hoger doorlooptijd heeft en een hoger rij waarde.

action_id	execute_time	explain_json	query_output	row_value
5ee202b1e09b25.01034714	0.063425064086914	{{"table": "partner protocol", "kev": "PRI..."	SELECT protocoltaak.id AS or...	83580
5ee202b1e09b25.01034714	0.0090811252593994	{{"table": "partner protocol", "kev": "PRI..."	SELECT protocoltaak.id AS or...	3902
5ee202b1e09b25.01034714	0.0020301342010498	{{"table": "protocoltaak koppeling", "kev": "PRI..."	SELECT protocoltaak.id AS or...	3558
5ee202b1e09b25.01034714	0.0019378662109375	{{"table": "protocoltaak koppeling", "kev": "PRI..."	SELECT protocoltaak.id AS or...	3226
5ee202b1e09b25.01034714	0.0012459754943848	{{"table": "v1", "kev": "reden ziekmelding..."	SELECT v2.id	F... 14

Afbeelding 10: Explain bevat filesort of temporary tables

8.5 Graylog

Een vervolgstap is om gebruik te maken van graylog zoals aangegeven in hoofdstuk [7.6.5 Graylog](#). Hier kunnen de waarden van MariaDb beter worden gevisualiseerd met grafieken en tabellen. Zo kan er op een oogopslag worden gezien of een bepaalde actie afwijkende query's hebben. Aangezien verzuimsignaal al gebruik maakt van graylog en dat de gemaakte applicatie de mogelijkheid biedt om dit toe te voegen is het een goede vervolgstap om meer duidelijkheid in destructive queries te creëren.

8.6 Testresultaten

Aan de hand van de requirements zijn testen gedaan in combinatie met de acceptatiecriteria van deze requirements.

Als eerste is er gekeken hoeveel vertraging de nieuwe koppeling heeft opgeleverd, door de tijd te meten van de nieuwe methode die is gerealiseerd. Uit deze meting is gekomen dat de methode minder dan 1 milliseconde tijd kost. Dit is een zodanig korte tijd dat het bijna niet merkbaar is dat de extra klasse DBlistener is toegevoerd. Ook is er gekeken naar het laden van de front-end. De pagina is vrijwel direct zichtbaar net zoals voorheen.

Verder is er ook een stresstest uitgevoerd op de applicatie door middel van DataExchange. DataExchange is een applicatie die koppeling legt tussen Verzuimsignaal en de klant. Bij een koppeling wordt er in een korte tijd veel query's uitgevoerd. In het geval van de test zijn er meer dan honderdduizend query's over de applicatie gelopen. Het resultaat was dat

de applicatie op dezelfde snelheid bleef doorlopen en dat de applicatie niet crashed onder de zware load.

Ook is er gekeken of de query's op een juiste manier in de database zijn aangekomen. Om dit te controleren is er gebruik gemaakt van van de honderdduizenden query's die via de vorige test in de database zijn gekomen. Bij het bekijken van de tabel van de database waren de kolommen naar wens gevuld. Op deze kolommen kunnen via database query's gegevens worden uitgefilterd en eventueel conclusie worden getrokken zoals in hoofdstuk [8.4 MariaDB](#).

Wanneer er gebruik wordt gemaakt van het filter van de applicatie worden alleen de aangegeven berichten in de database gezet zoals is aangegeven in hoofdstuk [8.3.1 Consumenten berichten](#).

Ook is er gekeken of de juist scripts/paginanaam in de database is gekomen onder de honderdduizenden query's zijn alle velden van deze kolom gevuld. Er is hier geen enkele waarde met de standaardwaarde "null" gevuld.

Verder is zichtbaar welke acties er zijn. Het id is ook bij dit veld overal gevuld. Het is zelfs inzichtelijk dat de koppelingen een actie is. Elke koppeling heeft een eigen "action_id" en er wordt weergegeven hoeveel query's binnen de actie zijn uitgevoerd. Zoals weergegeven in onderstaande afbeelding.

action_id	aantalActies
5ee202b1e09b25.01034714	115305
5ee207c43c33c9.06681085	74134
5ee1fd529b7d88.49170165	49086
5ee2078a46e290.64390251	31320

Afbeelding 11: weergaven van splitsing in action_id

Er is ook gebruik gemaakt van unit testing. Dit is op de message klasse gedaan van de Query detector manager. Aan de hand van de verschillende testen kan er snel worden gezien of de klasse functioneert naar behoren. Ook kunnen vroegtijdig bugs worden voorkomen wanneer er bijvoorbeeld aanpassingen worden gedaan in de klasse.

8.7 Resultaat

Het is mogelijk om query's van het Verzuimsignaal systeem te transporteren naar de query detector via RabbitMQ. Daarbij is de snelheid van de applicatie ongemoeid gebleven en is de koppeling bijna niet merkbaar. Door middel van de gegevens kunnen er conclusies uit de query's worden getrokken ([8.4 MariaDB](#)). Denk hier aan duplicaten of de grootte van een actie. Ook is het nu inzichtelijk wanneer een query een lange doorlooptijd heeft. Het prototype die is gemaakt moet nog eerst op de test- en acceptatieomgeving worden getest, hier is namelijk relevantere data. Ook is een goede vervolgstap om graylog te

integreren binnen het systeem. Op deze manier kan er een beter visueel beeld worden gegeven. De opdrachtgever heeft bij het prototype wel baat. Voorheen was er geen inzicht in de query's wat nu wel mogelijk is. Er kan nu in kaart worden gebracht wat het gedrag van de query's is. Ook zouden aan de hand van deze gegevens eventueel verbeteringen op het gebruiken van de database kunnen worden gedaan.

9. Slot

Bij het detecteren van destructive query's is er als eerst gekeken naar hoe query's kunnen worden afgevangen. Hier is gekozen om dit via de klasse DBConnector te doen van de verzuimsignaal applicatie. Aangezien over deze klasse alle query's gaan. Ook is hier de benodigde informatie voor het bepalen van een destructieve query. Vervolgens is er gekeken hoe de query's kunnen worden getransporteerd naar de Query detector. Hier is uiteindelijk gekozen voor RabbitMQ. Het grote voordeel hiervan is dat dit al reeds in de applicatie aanwezig is. Ook heeft het de benodigde eisen voor het transporteren. Verder is er gekeken wat een query definieert. Hier is uitgekomen dat het belangrijk is dat een query op doorlooptijd en duplicaten moet worden bekeken. Ook is er gekeken naar het aantal rijen van de database die worden aangeraakt bij het uitvoeren van een query. Voor het analyseren van een query is er gebruik gemaakt van "Explain". Dit is een eigenschap die kan worden meegegeven aan een query om meer informatie te krijgen over de uitgevoerde query. Tot slot is er bepaald met de opdrachtgever wat er gemonitord moet worden. Hier is uitgekomen om de duplicaten, doorlooptijd, aantal rijen en het aantal queries dat binnen een actie wordt uitgevoerd.

Als volgende stap is er een ontwerp gecreëerd. Dit is aan de hand van verschillende requirements en diagrammen gedaan. Ook is het inzichtelijk gemaakt hoe de uitgevoerde queries er uit moeten gaan zien.

In de realisatie is er een prototype gerealiseerd. Dit prototype geeft antwoord op de hoofdvraag (hoe kunnen destructieve query worden gedetecteerd). Aan de hand van het ontwerp is dit prototype gerealiseerd. Het is hierdoor inzichtelijk geworden hoeveel dubbele query's er worden uitgevoerd, welke query's het langst duren en ook is het inzichtelijk hoeveel queries er worden uitgevoerd binnen een actie.

Een goede vervolgstap voor dit prototype is om gebruik te maken van graylog. Hiermee kan het resultaat visueel inzichtelijk worden gemaakt aan de hand van grafieken. Hierdoor kan er in een oogopslag worden weergegeven wanneer een query/actie rare verschijnen vertoont.

Wanneer dit is gecreëerd kan de applicatie worden verplaatst naar de test- en/of acceptatieomgeving. Hier kan de applicatie op een meer realistische omgeving worden getest. Aan de hand van de resultaten kunnen er eventueel verbetering aan de database/applicatie worden gedaan. Denk hier aan het verminderen van duplicaten of indexes toevoegen waar nodig.

Literatuurlijst

- [1] Aggarwal S. (2018). *RabbitMQ vs Apache Kafka – Linux Hint*. Retrieved from <https://linuxhint.com/rabbitmq-vs-apache-kafka/>
- [2] HBO-I; Methodes. (n.d.). Retrieved from <http://ictresearchmethods.nl/Methods>
- [3] Johansson L. (2019). *Part 1: RabbitMQ Best Practices - CloudAMQP*. Retrieved from <https://www.cloudamqp.com/blog/2017-12-29-part1-rabbitmq-best-practice.html>
- [4] Levy E. (2019). *Kafka vs. RabbitMQ: Architecture, Performance & Use Cases*. Retrieved from <https://www.upsolver.com/blog/kafka-versus-rabbitmq-architecture-performance-use-case>
- [5] *Using EXPLAIN to Write Better MySQL Queries — SitePoint*. (2012, 4 4). Retrieved from Shameer C.: <https://www.sitepoint.com/using-explain-to-write-better-mysql-queries/>
- [6] *zeromq*. (2015, januari 10). Retrieved from Brave new geek: <https://bravenewgeek.com/tag/zeromq/>

Bijlagen

Bijlage 1: Onderzoeksverslag

verzuimsignaal



In de bijlage is het volledige onderzoek terug te lezen. Hier staan onder andere in hoe te transporteren, hoe te analyseren, wat is een destructive query en wat is belangrijk bij monitoring.

Bijlage 2: Technisch Ontwerp



In de Bijlage zijn alle ontwerpen en requirements gedetailleerd uitgelegd. Het document bevat onder andere een component diagram en meerdere klassediagrammen