

Client-side evaluatie van GeoSPARQL opvragingen over heterogene gegevensbronnen

Andreas De Witte

Studentennummer: 01407414

Promotoren: dr. ing. Pieter Colpaert, dr. ir. Ruben Taelman

Begeleiding: Brecht Van de Vyvere, Julian Andres Rojas Melendez

Masterproef ingediend tot het behalen van de academische graad van Master of Science in de Industriële Wetenschappen: informatica

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. Bart Dhoedt
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2019-2020



Dankwoord

Na zes maanden werken is deze masterproef afgerond. Ik heb hier zeer veel aan gewerkt en zonder de hulp van verschillende personen zou het mij niet gelukt zijn.

Allereerst wil ik mijn promotor Pieter Colpaert bedanken voor de continue begeleiding en feedback. Daarnaast wil ik Pieter ook bedanken voor de motivatie en alle opportuniteiten die hij mij gegeven heeft.

Graag zou ik ook mijn tweede promotor Ruben Taelman bedanken voor de vele technische uitleg en de veelvuldige feedback die mijn thesis tot een goed einde gebracht heeft.

Daarnaast wil ik ook graag Ruben Dedecker bedanken voor de tips over het schrijven, de gezamenlijke debugsessies en de raad over het opzetten van de demonstraties.

Tevens zou ik graag mijn medestudenten Demian Dekoninck en Thomas Aelbrecht bedanken voor de steun bij het maken van deze masterproef. Ook van jullie heb ik zeer veel motivatie gekregen, alsook heeft jullie input verschillende van mijn problemen opgelost.

Ook zou ik graag mijn familie en vrienden bedanken om altijd klaar te staan voor mij in deze zware periode. Jullie zorgden zo nu en dan voor de nodige afleiding en ontspanning na een dag werken. Hierbij gaven jullie mij telkens de nodige aanmoediging om door te gaan.

Tot slot zou ik iedereen willen bedanken die mij geïnspireerd heeft tijdens het schrijven en iedereen die mijn thesis heeft nagelezen.

Bedankt iedereen!

Andreas De Witte

Client-side evaluatie van GeoSPARQL opvragingen over heterogene gegevensbronnen

Andreas De Witte

Supervisor(s): dr. ing. Pieter Colpaert, dr. ir. Ruben Taelman, Brecht Van de Vyvere, Julian Andres Rojas Melendez

Abstract—Op het Web, zoals het nu gekend is, kunnen gebruikers gemakkelijk pagina's van websites begrijpen. Voor computers is dit echter niet het geval, hier moet enorm veel moeite gedaan worden om betekenis en context uit de zinnen te ontleiden. Het Web zoals het nu is, is niet gebouwd om begrepen te worden door machines. Dankzij het Semantische Web, wat een uitbreiding op het huidige Web is, is het wel mogelijk voor machines om de pagina's van websites te begrijpen.

Momenteel is het al mogelijk om opzoeken naar gelinkte data te doen over een beperkt aantal gegevensbronnen, omdat weinig gegevensbronnen gelinkte data ondersteunen. Deze gegevensbronnen kunnen ook heterogeen zijn, wat betekent dat het andere types van bronnen kunnen zijn. Deze opzoeken gebeuren aan de hand van SPARQL, waarvan verschillende werkende implementaties bestaan. Om geografische opvragingen te doen wordt GeoSPARQL gebruikt. Hiervan zijn echter weinig implementaties gemaakt en deze implementaties hebben in vele gevallen incorrecte gedragingen.

In dit werk is een beperkte implementatie van GeoSPARQL gemaakt om zo de informatie in RDF-formaat op te halen en topologische relaties uit te rekenen. Hierbij is getest bij welk soorten interfaces deze topologische relaties op de client kunnen worden berekend. Dit berekenen op de client is belangrijk voor vele redenen. Eén van de belangrijkste redenen is dat deze berekeningen een server zouden kunnen verlammen, terwijl deze berekeningen voor slechts één gebruiker op de client zeer goed mogelijk zijn. In andere woorden, dit op de client doen is een effectieve manier om de belasting te verspreiden. Deze paper geeft nieuwe inzichten over het afhandelen van deze geografische opvragingen op de client.

Zo blijkt dat het zeer eenvoudig is om topologische relaties te berekenen op de client wanneer de bron een “data dump” is. Hierbij zal de client de volledige dataset moeten downloaden. Daarnaast is het mogelijk om de topologische relaties te berekenen wanneer de bron een “TPF interface” is. Deze zal zelf al optimalisaties voorzien door enkel de benodigde gegevens terug te geven. Wanneer de bron echter een SPARQL-endpoint is, is dit moeilijker. Dit wordt mogelijk gemaakt door de verschillende RDF-triples te overlopen en te tellen hoeveel resultaten overeenkomen. Hierbij kan zo het kleinste patroon gevormd worden om te voorkomen dat meer data opgehaald wordt dan noodzakelijk is.

Zo kan geconcludeerd worden dat het afhandelen van deze opvragingen veel beter op de client gedaan kan worden. Zo kan het geheel van de opvraging weergegeven worden, zelfs wanneer de bron dit zelf niet ondersteunt. Deze masterproef is vooral nuttig voor computerwetenschappers die echte experts zijn van het Semantische Web, maar kan ook gebruikt worden door geïnteresseerden voor het verkrijgen van een betere kennis van het Semantische Web en zijn mogelijkheden.

Keywords— Semantisch Web, gelinkte data, OGC, GeoSPARQL, client-side, topologische relatie

I. INLEIDING

Het Web is gemaakt om begrepen te worden door mensen. Dit zorgt ervoor dat het voor machines veel moeilijker is om het Web te interpreteren. Zo zijn simpele taken zeer moeilijk te automatiseren. Een voorbeeld hiervan is de planning van een daguitstap. Hierbij zou het de bedoeling zijn dat een *intelligent agent* volledig autonoom zou kunnen inplannen welke uitstap gemaakt wordt, rekening houdend met de deelnemers, het weer, interesses,...

Het Web heeft ook nog andere problemen. Zo worden zeer veel gegevens over personen bijgehouden door grote bedrijven (zoals Google, Facebook,...), terwijl dit eigenlijk beter beheerd kan worden door de persoon zelf. Zo kan de persoon zelf beslissen wat hij vrij wil geven en wat niet. Dit zou gebeuren aan de hand van gelinkte data, wat verder besproken wordt.

Deze uitbreiding op het Web heet het Semantische Web. Dit Web maakt gebruik van gelinkte data, wat dan weer opgehaald kan worden met behulp van SPARQL (een zoektaal). Een uitbreiding bovenop SPARQL is GeoSPARQL. Dit alles wordt verderop uitgebreider besproken.

Dit artikel legt de focus op de uitbreiding van verschillende “Linked Data publicatie”-interfaces. Hierbij is het de bedoeling om deze interfaces uit te breiden met GeoSPARQL-functionaliteiten door de filtering op de client uit te voeren.

II. LITERATUURSTUDIE

Het Semantische Web is een Web dat geïnterpreteerd kan worden door zowel mensen als machines. Om het Semantische Web te creëren zijn verschillende stappen nodig. Hierbij is betekenis belangrijk voor computers. Daarnaast moet dit gerepresenteerd kunnen worden voor machines, wat gebeurt aan de hand van RDF (= *Resource Description Framework*). Dit wordt beide aangepakt met behulp van *Linked Data*. Ook is decentralisatie een belangrijk aspect van het Semantische Web. Dit betekent dat er geen centrale eenheid mag zijn die alle informatie bijhoudt, maar dat er meerdere kleine entiteiten zijn die elk een deel bijhouden [1].

Naast het plaatsen van data op het Web heeft het Semantische Web nog een ander (en belangrijker) doel, namelijk het maken van links. Wanneer er data verkregen zijn, bevatten deze links naar waar gerelateerde data te vinden zijn. Op deze manier krijgen de data meer context. Bij deze data is het belangrijk dat ze open en toegankelijk zijn

om herbruikt te worden [2].

Het RDF voorziet een algemene methode om relaties tussen data objecten te beschrijven. Zo is RDF vooral efficiënt om informatie van verschillende bronnen te integreren door de informatie los te koppelen van zijn schema. Bij RDF wordt gebruik gemaakt van triples van de vorm *subject - predicate - object* [3]. Het is wel belangrijk te weten dat RDF geen dataformaat is, maar een datamodel. Dit betekent dat de data eerst geserialiseerd moeten worden voordat deze gepubliceerd kunnen worden. De meest gebruikte RDF-syntaxen zijn: “RDF/XML”, “RDFa”, “Turtle”, “N-Triples” en “JSON-LD”. Hierbij is “Turtle” de meest compacte en is “JSON-LD” de meest voorkomende (vanwege de overeenkomsten met het bekende JSON formaat) [4].

SPARQL is een zoektaal voor de opzoeking van RDF gebaseerde gegevens. Deze taal heeft meerdere gelijkenissen met SQL. SPARQL is dus eigenlijk een query-taal die gebruikt wordt om gegevens op te halen van het Web. Het belang van SPARQL voor deze masterproef is dat er een werkende implementatie van SPARQL vereist is om een implementatie van GeoSPARQL te maken, aangezien GeoSPARQL een uitbreiding is van SPARQL [5].

Comunica is een modulaire SPARQL *query engine* voor het Web, gemaakt door het IDLAB van de universiteit Gent. Comunica is de werkende implementatie van SPARQL waarvan vertrokken wordt. Hierbij is gekozen voor Comunica omdat deze ontwikkeld is met vijf zeer specifieke doelen [6]:

1. Het moet SPARQL queries kunnen evalueren;
2. Het moet modulaair zijn;
3. Het moet over heterogene interfaces kunnen queryen;
4. Het moet gefedereerd (= naar meerdere bronnen tegelijkertijd) kunnen queryen;
5. Het moet gemaakt zijn met webtechnologieën.

Het OGC (= Open Geospatial Consortium) is een wereldwijde *community* die poogt om de manier waarop omgegaan wordt met geospatiale locatie informatie te verbeteren. Om dit doel te bereiken, voorziet het OGC standaarden zoals “GML” en “WKT” voor de beschrijving van geografische objecten. Ook GeoSPARQL is een OGC standaard [7].

GeoSPARQL is één van de vele OGC standaarden en is uitermate geschikt voor het uitvoeren van GIS (= Geografische Informatie Systemen) queries. Maar dit is niet de enige mogelijkheid. GeoSPARQL gebruikt RDF en is een uitbreiding bovenop SPARQL. Het brengt een vocabulair voor het representeren van geospatiale data. Hiervoor hanteert het een architectuur (zie Figuur 1) die één hoofd-klasse bevat, zijnde “SpatialObject”. Hierbij zijn er twee andere klassen die hiervan overerven, namelijk “Feature” en “Geometry”. Het is belangrijk te weten dat een “Feature” object een “Geometry” object moet bevatten. Verder is een “Geometry” voorgesteld aan de hand van een “GML literal” of een “WKT literal” [7].

Bij GeoSPARQL is het belangrijk dat er een onderscheid wordt gemaakt tussen topologische functies en niet-topologische functies. De topologische functie beschrijven de relaties tussen objecten (bijvoorbeeld of een object in

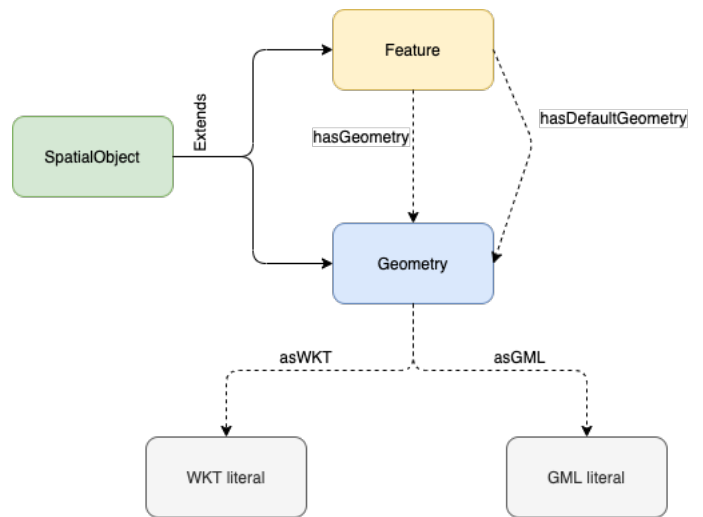


Fig. 1
VEREENVOUDIGD DIAGRAM VAN DE GEOSPARQL KLASSEN
“FEATURE” EN “GEOMETRY” MET SOMMIGE PROPERTIES (FIGUUR
GEBASEERD OP [8]).

een ander object ligt), terwijl de niet-topologische functies gevarieerder zijn (bijvoorbeeld de afstand tussen twee objecten geven).

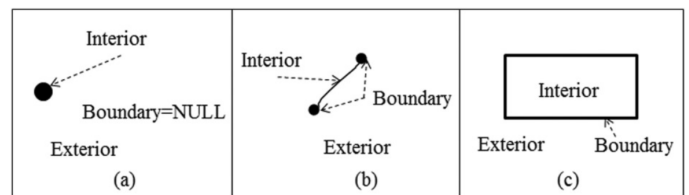


Fig. 2
SPATIALE OBJECTEN MET HUN *interior*, *boundary* EN *exterior*: (A)
EEN PUNT; (B) EEN LIJN; (C) EEN VLAk. FIGUUR VAN [9].

Voor de beschrijving van topologische relaties gebruikt GeoSPARQL het DE-9IM (= *Dimensionally Extended Nine-Intersection*) model. Dit model is een 3x3 matrix waarbij de *interior*, *boundary* en *exterior* van het ene spatiale object vergeleken wordt met deze van het andere spatiale object. De betekenis van *interior*, *boundary* en *exterior* is verduidelijkt in Figuur 2. De 3x3 matrix voor twee objecten “a” en “b” is van de volgende vorm [9]:

$$DE - 9IM(a, b) = \begin{bmatrix} \dim(I(a) \cap I(b)) & \dim(I(a) \cap B(b)) & \dim(I(a) \cap E(b)) \\ \dim(B(a) \cap I(b)) & \dim(B(a) \cap B(b)) & \dim(B(a) \cap E(b)) \\ \dim(E(a) \cap I(b)) & \dim(E(a) \cap B(b)) & \dim(E(a) \cap E(b)) \end{bmatrix}$$

Ten slotte voorziet GeoSPARQL ook een functionaliteit voor het herschrijven van een query. Deze is nodig wanneer een topologische functie gebruikt wordt als predicaat. Dit is nodig omdat dit enkel als predicaat kan bestaan wanneer dit al uitgerekend is op de server. Dit wordt niet verder

aangehaald in deze masterproef, aangezien de filtering op de client gebeurt, dus kan er niet uitgegaan worden van berekeningen op de server. Een ander argument om niet uit te gaan van deze berekening op de server is dat de data van verschillende bronnen kunnen komen. Dit is dus niet bruikbaar om a priori door één bron uitgerekend worden [7].

III. IMPLEMENTATIE

Bij de implementatie van de GeoSPARQL- functionaliteiten is gebruik gemaakt van Comunica. Hier is een actor aangemaakt die de GeoSPARQL *query engine* zal instantiëren. Hiervoor wordt gebruik gemaakt van de *library* “sparqlalgebra” om de SPARQL query om te vormen naar SPARQL algebra. Bovendien wordt “sparqlee” gebruikt om deze SPARQL algebra correct uit te voeren. Met andere woorden, “sparqlee” is een *expression evaluator*. Bij deze implementatie zijn de effectieve GeoSPARQL- functionaliteiten dus geïmplementeerd binnen “sparqlee”.

Om de datastructuur van GeoSPARQL te respecteren (zie Figuur 1) is gebruik gemaakt van GeoJSON. Dit is een vaker gebruikt formaat dat ondersteuning biedt voor zowel “Geometry” als “Feature” objecten. Om nu van een “WKT literal” naar GeoJSON te kunnen overgaan, wordt gebruik gemaakt van Terraformer.

Het volgende probleem is het oplossen van de topologische- en niet-topologische functies. Hiervoor gebruikt men “Turf.js”. Turf voorziet vele methoden die gebruikt kunnen worden om berekeningen te doen met geospatiale objecten. Daarnaast heeft Turf ook enkele *build in* functies (zoals “booleanContains”) die rechtstreeks gebruikt kunnen worden. Naast het functionele is Turf zo een goede keuze omdat het een grote *community* heeft, waardoor bugs snel gedetecteerd worden. Bovendien is Turf modulaair geprogrammeerd, waardoor slechts de kleine modules die benodigd zijn, ingeladen moeten worden.

Het laatste probleem om een werkende implementatie van GeoSPARQL te bekomen, is het probleem van de verschillende projecties. Dit wordt opgelost dankzij “Proj4js”. Proj4 is een *library* die zorgt voor de transformatie van coördinaten in het ene referentiesysteem naar coördinaten van het andere referentiesysteem.

Nu een werkende implementatie van GeoSPARQL ter beschikking is, blijft nog één implementatie onafgewerkt. Dit is een testomgeving voor het controleren welke “Linked Data publicatie”-interfaces uitgebreid kunnen worden met GeoSPARQL- functionaliteiten. Hiervoor is gebruik gemaakt van de “jQuery Widget” van Comunica. Deze voorziet een grafische interface, waarbij het mogelijk is om de query voortdurend aan te passen, alsook de bronnen die gebruikt worden bij deze query. Ook zal deze grafische interface zowel het bekomen resultaat visualiseren als logs weergeven. Deze logs is het meest interessante deel om deze masterproef af te toetsen. Aan de hand van deze logs kan gecontroleerd worden hoe alles intern in zijn werk gaat.

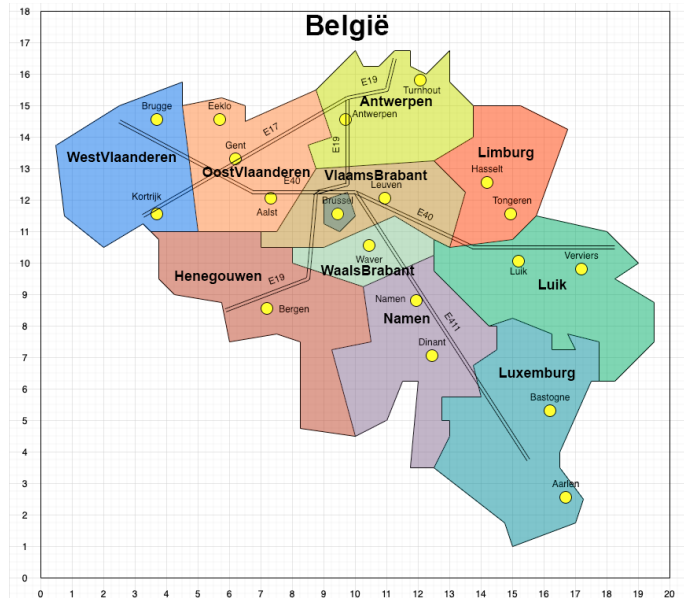


Fig. 3
TESTSET VOOR HET TESTEN VAN DE VERSCHILLENDE BRONNEN.

IV. INTERFACES

Alvorens de verschillende interfaces getest kunnen worden is het nodig om een gepaste use case te hebben. Vanwege de bekendheid is ervoor gekozen om België als use case te nemen voor het testen. Hierbij is een oppervlakige tekening van België gemaakt, die te zien is in Figuur 3. Deze tekening is gemaakt in een aangepaste schaal. Verder is deze tekening identiek (op vlak van coördinaten) aan de bijhorende dataset, die terug te vinden is op GitHub Gist¹. Deze dataset is opgesplitst in vijf verschillende bestanden (namelijk: “land”, “gewest”, “provincie”, “weg” en “stad”) zodat deze dataset bruikbaar is voor het verifiëren dat gefedereerd queryen nog steeds mogelijk is.

De eerste bron die gecontroleerd wordt is de “data dump”. Deze wordt ook gebruikt als baseline, omdat dit de eenvoudigste vorm is. Bij de “data dump” wordt de volledige dataset gedownload op de client. Op deze manier kan de client de filtering volledig uitvoeren.

De tweede bron is de “TPF interface”. Hierbij zal een server de bestanden met data beheer en antwoorden op aanvragen. Bij het queryen zal de query opgedeeld worden in verschillende *triple pattern fragments*, zodat de server zelf kan beslissen om geen overbodige informatie mee te sturen. Vervolgens zal de client deze gegevens joinen, zodat hij uiteindelijk het geheel kan filteren om zo de correcte resultaten te kunnen weergeven.

De derde bron is het “SPARQL endpoint”. Deze is met zekerheid de moeilijkste. Deze bron heeft zelf de mogelijkheid om SPARQL queries op te lossen, maar deze ondersteunt zelf geen GeoSPARQL. Het is dus niet mogelijk om de GeoSPARQL query in zijn geheel door te sturen naar het “SPARQL endpoint” (bovendien zou gefedereerd

¹<https://gist.github.com/dreeki/e48bbe533a4b1191045b3652ff2c9c81>

queryen niet mogelijk zijn moest het “SPARQL endpoint” de query volledig zelf uitwerken). Dit probleem wordt aangepakt door de query te overlopen over zijn individuele RDF-triples. Zo zal de *query engine* eerst een “count” query maken om te weten waar het kleinste mogelijke patroon is. Dit is nodig om een goede performantie te bekomen. De tweede stap is het effectief ophalen van dit kleinste patroon. Dit wordt meerdere malen herhaald tot de volledige query verwerkt is. Zo wordt het geheel opnieuw gejoined op de client, zodat hier wederom de filtering kan gebeuren.

V. CONCLUSIE

Om te antwoorden op de vraag welke “Linked Data publicatie”-interfaces uitgebreid kunnen worden met GeoSPARQL-functionaliteiten door de filtering op de client uit te voeren, moeten de resultaten van voorheen geïnterpreteerd worden.

Zo is het mogelijk om dit te doen bij “data dumps” doordat de client de volledige dataset downloadt en deze vervolgens filtert. Bij “TPF interfaces” is dit ook mogelijk, door de verschillende *triple pattern fragments* te joinen op de client. Dit resultaat kan zo ook weer gefilterd worden. Zelfs bij een “SPARQL endpoint” is dit mogelijk door bij de bron te tellen hoeveel resultaten er zijn voor elk RDF-triple van de query. Hierbij wordt het kleinste patroon opgehaald en gejoined op de client. Ook dit resultaat wordt opnieuw gefilterd op de client.

REFERENCES

- [1] Berners-Lee, Tim and Hendler, James and Lassila, Ora *The semantic web*, Scientific american, vol. 284, no. 5, pp. 3443, 2001.
- [2] Berners Lee, Tim *Linked Data*, 2006.
- [3] Lassila, Ora and Swick, Ralph R and others *Resource description framework (RDF) model and syntax specification*, 1998.
- [4] Heath, Tom and Bizer, Christian *Linked data: Evolving the web into a global data space*, Synthesis lectures on the semantic web: theory and technology, vol. 1, no. 1, pp. 1136, 2011.
- [5] Harris, Steve and Seaborne, Andy *SPARQL 1.1 Query Language*, World Wide Web Consortium, 2013.
- [6] Taelman, Ruben and Van Herwegen, Joachim and Vander Sande, Miel and Verborgh, Ruben *Comunica: a modular SPARQL query engine for the web*, in International Semantic Web Conference. Springer, 2018, pp. 239255.
- [7] *Open Geospatial Consortium*, URL: <https://ogc.org>
- [8] *GeoSPARQL support: What is GeoSPARQL*, URL: <http://graphdb.ontotext.com/documentation/standard/geosparql-support.html>
- [9] Shen, Jingwei and Chen, Min and Liu, Xintao *Classification of topological relations between spatial objects in two-dimensional space within the dimensionally extended 9-intersection model*, Transactions in GIS, vol. 22, no. 2, pp. 514541, 2018.

Client-side evaluation of GeoSPARQL queries over heterogeneous data sources

Andreas De Witte

Supervisor(s): dr. ing. Pieter Colpaert, dr. ir. Ruben Taelman, Brecht Van de Vyvere, Julian Andres Rojas Melendez

Abstract— On the Web as it's currently known, users can easily understand pages of websites. This is not the case for computers, as they need to do a lot of effort to extract both meaning and context from sentences. The Web, as it is today, is not built to be interpreted by machines. Thanks to the Semantic Web, which is an extension to the current Web, it is possible for machines to understand the pages of websites.

It is currently possible to query for linked data to a limited amount of data sources, because very few data sources support linked data. These data sources can be heterogeneous, which means they can be different types of data sources. These queries are executed with SPARQL, which has multiple working implementations. For geographical queries, GeoSPARQL is needed. However, there are only few implementations of GeoSPARQL and most of these implementations are working incorrectly.

In this work, a limited implementation of GeoSPARQL is made in order to request data in RDF format and calculate the topological relations in this data. With this implementation, it has been tested for which interfaces these topological relations can be calculated on the client-side. Doing this on the client-side is important for many reasons. The major reason is that many of these calculations would cripple a server, while these calculations for only one user on the client-side is a more feasible solution. In other words, doing this on the client-side is an effective way of spreading the load. This paper gives new insights about handling geographical queries on the client-side.

Like this, it seems to be very simple to calculate the topological relations on the client-side when the data source is a data dump. Hereby, the client will have to download the entire data set. It's also possible to calculate topological relations when the source is a TPF interface. The interface will already provide several optimizations by only returning the necessary data. However, when the source is a SPARQL endpoint, this is more difficult. This is possible by iterating over the different RDF triples and by counting the amount of matching results. Like this, the smallest pattern can be formed and retrieved from the SPARQL endpoint. This prevents the retrieval of unnecessary data.

The conclusion can be made that these kind of queries can be handled better on the client-side. Like this, the entire query can be processed, even when the source doesn't fully support it. This master's thesis is mostly useful for computer scientists who are true experts about Semantic Web, but it can also be used by enthusiasts who want to receive a better understanding of the Semantic Web and it's possibilities.

Keywords— Semantic Web, linked data, OGC, GeoSPARQL, client-side, topological relation

I. INTRODUCTION

The Web is made to be understood by humans. However, this makes it harder for machines to interpret the Web. Because of this, simple tasks are very hard to automate. An example of this is planning a daytrip. The idea with this

would be that an intelligent agent would be fully capable of autonomously planning the daytrip, keeping in mind the participants, the weather, common interests,...

The Web has some other flaws too. For example, bigger companies (like Google, Facebook,...) are collecting huge amounts of data about people, while this should rather be managed by the people themselves. Like this, the person himself would be able to decide which data he wants to share and more importantly, which data he doesn't want to share. This would be possible using Linked Data, which is discussed later on.

This extension to the Web is called the Semantic Web. This Web uses Linked Data, which is queried using SPARQL (a query language). GeoSPARQL is an extension of SPARQL. This as well will be discussed more broadly, later on.

This article focusses on the extension of the different "Linked Data publication" interfaces. Hereby, the goal is to extend these interfaces with GeoSPARQL functionalities by executing the filtering on the client.

II. STATE OF THE ART

The Semantic Web is a Web that can be interpreted by both humans and machines. To make the Semantic Web a reality, several steps are needed. Hereby, meaning is important for computers. Also, this has to be represented in a way machines can understand, which is done with RDF (= Resource Description Framework). Both these problems are tackled with Linked Data. Another important aspect of the Semantic Web is decentralization. This means that there can not be a single entity that holds all the data, but instead many small entities that each hold a little part of the data [1].

Apart from posting data on the Web, the Semantic Web has another (and even more important) goal, being the creation of links. When data is retrieved, this data should contain links to where related data can be found. By doing so, the data gets more context. It's important that this data is both open and accessible to be reused [1].

RDF provides a general method to describe relations between data objects. This makes RDF most efficient to integrate information from multiple sources, by decoupling the information from it's scheme. RDF uses triples, which have the form "subject - predicate - object" [3]. It is important to know that RDF is not a data format, but a data model. This means that the data must be serialized before it can be published. The most used RDF syntaxes are:

“RDF/XML”, “RDFa”, “Turtle”, “N-Triples” and “JSON-LD”. In these formats, “Turtle” is the most compact one and “JSON-LD” is the most used one (because of it’s similarities with the well known JSON format) [4].

SPARQL is a query language for querying RDF based data. This language has many similarities with SQL. SPARQL is actually a query language, used for retrieving data from the Web. The importance of SPARQL for this master’s thesis is that a GeoSPARQL implementation requires a working SPARQL implementation, because GeoSPARQL is an extension of SPARQL [5].

Comunica is a modular SPARQL query engine for the Web, made by the IDLAB of the university of Ghent. Comunica is the working implementation of SPARQL that is used as a start. Hereby, Comunica is chosen because it was developed with five specific goals [6]:

1. It must be able to evaluate SPARQL queries;
2. It must be modular;
3. It must be able to query over heterogeneous interfaces;
4. It must be able of querying federated (= multiple sources at once);
5. It must be made using web technologies.

The OGC (= Open Geospatial Consortium) is a worldwide community that tries to improve the way how geospatial location information is handled. To achieve its goal, the OGC provides standards like “GML” and “WKT” for the description of geographical objects. As well, GeoSPARQL is an OGC standard [7].

GeoSPARQL is one of the many OGC standards and is mostly fit for the execution of GIS (= geographical Information System) queries. However, this is not its only possibility. GeoSPARQL uses RDF and is an extension on SPARQL. It brings a new vocabulary for representing geospatial data. To do so, it uses an architecture (see Figure 1) which contains one main class, being “SpatialObject”. The other two classes, “Feature” and “Geometry”, are both inherited from the “SpatialObject” class. It’s important to know that a “Feature” object must contain a “Geometry” object. A “Geometry” is represented by a “GML literal” or a “WKT literal” [7].

With GeoSPARQL, it’s important to distinguish the topological functions from the non-topological functions. Topological functions describe relations between objects (for example whether an object lies inside another object or not), while non-topological functions are more varied (for example returning the distance between two objects).

In order to describe topological relation using GeoSPARQL, the DE-9IM (= Dimensionally Extended Nine-Intersection) model is used. This model is a 3x3 matrix that compares the interior, boundary and exterior of two spatial objects. The meaning of interior, boundary and exterior are clarified in Figure 2. The 3x3 matrix for two objects “a” and “b” has the following form [9]:

$$DE - 9IM(a, b) =$$

$$\begin{bmatrix} \dim(I(a) \cap I(b)) & \dim(I(a) \cap B(b)) & \dim(I(a) \cap E(b)) \\ \dim(B(a) \cap I(b)) & \dim(B(a) \cap B(b)) & \dim(B(a) \cap E(b)) \\ \dim(E(a) \cap I(b)) & \dim(E(a) \cap B(b)) & \dim(E(a) \cap E(b)) \end{bmatrix}$$

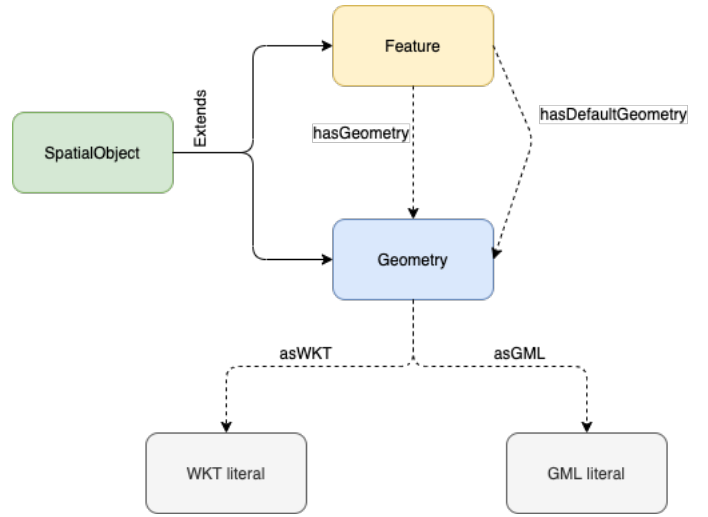


Fig. 1

SIMPLIFIED DIAGRAM OF THE GEOSPARKL CLASSES “FEATURE” AND “GEOMETRY” WITH SOME PROPERTIES (FIGURE BASED ON [8]).

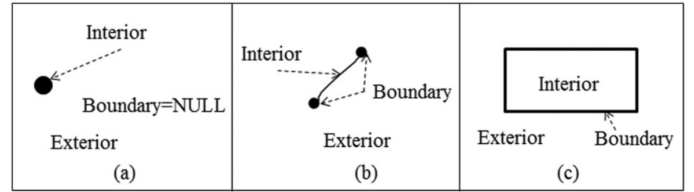


Fig. 2

SPATIAL OBJECTS WITH THEIR INTERIOR, BOUNDARY AND EXTERIOR: (A) A POINT; (B) A LINE; (C) A POLYGON. FIGURE FROM [9]

Last but not least, GeoSPARQL brings a functionality to rewrite queries. This must be done when a topological function is used as a predicate. This is needed because it can only exist as a predicate when this has been pre computed on the server. This is not mentioned again in this master’s thesis since the filtering is done on the client, hence assumptions of precomputations on the server cannot be made. Another argument to not assume precomputations on the server is that data can originate from multiple sources. This means it’s not useful to precompute this on the server [7].

III. IMPLEMENTATION

Comunica is used for the implementation of the GeoSPARQL functionalities. In Comunica, an actor is created who initializes a GeoSPARQL query engine. This actor uses “sparqlalgebrajs” in order to transform the SPARQL query into SPARQL algebra. Moreover, “sparqllee” is used to have a correct execution of the SPARQL algebra. In other words, “sparqllee” is an expression evaluator. With this implementation, the GeoSPARQL functionalities are implemented within “sparqllee”.

To respect the datastructure of GeoSPARQL (see Figure 1), GeoJSON is used. This is a format that's already widely used and supports both "Geometry" and "Feature" objects. In order to turn the "WKT literal" into GeoJSON, Terraformer is used.

The next problem that needs to be solved, is executing both topological functions and non-topological function. For this, "Turf.js" is used. Turf provides many methods that can be used to do calculations with geospatial objects. Besides, Turf has some built in functions (like "boolean-Contains") that are immediately usable. In addition to its functional strength, Turf is a good choice because of its huge community. This community makes sure bugs are detected as soon as possible. Also, Turf is developed modularly, so only the small modules that are needed, have to be loaded.

The last thing to do to achieve a working implementation of GeoSPARQL is solve the problem of the different projections. This is solved thanks to "Proj4js". Proj4 is a library that takes care of the transformation from coordinates in one reference system to coordinates in another reference system.

Now, a working implementation of GeoSPARQL is available. Only one last implementation remains. This is a test environment to check which "Linked Data publication" interfaces can be extended with GeoSPARQL functionalities. To solve this, Comunica's "jQuery Widget" is used. This widget provides a graphical interface which enables the possibility to edit both the query and the sources used to execute the query. This graphical interface will also visualize the result of the query and it will show logging. This logging is mostly interesting to test the different "Linked Data publication" interfaces. Thanks to the logging, it's possible to check how the program works internally.

IV. INTERFACES

Before the different interfaces can be tested, an appropriate use case is needed. Because of it being well known, Belgium has been chosen as use case for the tests. Hereby, a shallow drawing of Belgium has been made, which is visualized in Figure 3. This drawing is made on a custom scale. Furthermore, this drawing is identical (in terms of coordinates) to the data set, which can be found on GitHub Gist¹. This data set is split into five different files (being: "land", "gewest", "provincie", "weg" and "stad". This respectively means "country", "region", "province", "road" and "city") in order to verify that federated querying is still possible.

The first source that needs to be checked is the "data dump". This will also be used as a baseline because this is the easiest source. With the "data dump", the entirety of the data set is downloaded onto the client. By doing so, the client can fully execute the filtering.

The second source is the "TPF interface". Hereby, the server will manage the files with data and respond to requests. While querying, the query itself will be divided into multiple triple pattern fragments, so the server itself can

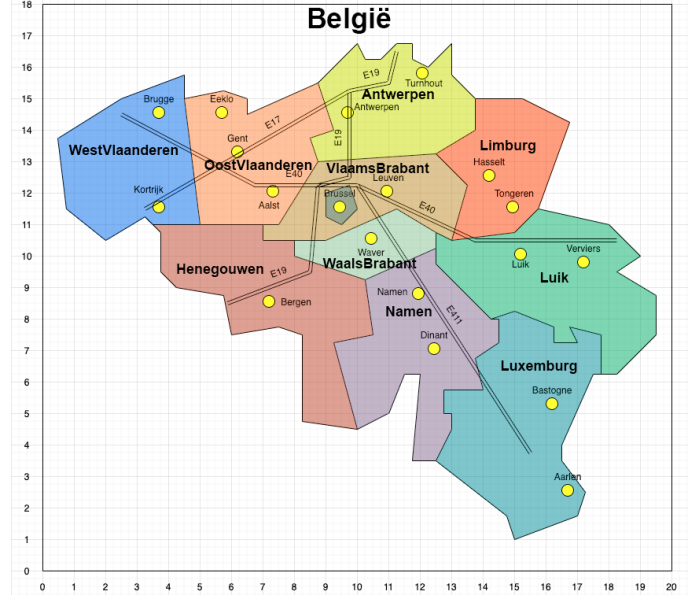


Fig. 3

TEST SET TO TEST DIFFERENT SOURCES.

decide not to send overflowing information. Afterwards, the client will join these results so it can filter the total result in the end. Like this, the correct result can be retrieved.

The third and final source is the "SPARQL endpoint". This one is surely the hardest. This source has the possibility of executing SPARQL queries on its own, but it doesn't support GeoSPARQL. This means that it's impossible to send the GeoSPARQL query as a whole to the "SPARQL endpoint" (moreover, federated querying would be impossible if the "SPARQL endpoint" were to fully execute the query on its own). This problem is tackled by iterating over the query its individual RDF triples. Like this, the query engine will first send a "count" query in order to know what the smallest pattern is. This is needed to achieve a good performance. The second step is the actual retrieval of this smallest pattern. This is repeated multiple times, until the entire query is processed. In the end, the entirety of results is joined on the client, so the client is capable of executing the filtering once again.

V. CONCLUSIE

To formulate an answer to the question which "Linked Data publication" interfaces can be extended with GeoSPARQL functionalities by filtering on the client, the earlier results have to be interpreted.

This is possible for "data dumps" because the client downloads the entire data set and filters afterwards. For "TPF interfaces", this is also possible by joining the different triple pattern fragments on the client. The result can be filtered like this once again. Even with "SPARQL endpoints", it is possible by counting at the source how many results there are for each RDF triple in the query. Hereby, the smallest patterns are retrieved and joined on the client.

¹<https://gist.github.com/dreeki/e48bbe533a4b1191045b3652ff2c9c81>

This result is also filtered on the client.

REFERENCES

- [1] Berners-Lee, Tim and Hendler, James and Lassila, Ora *The semantic web*, Scientific american, vol. 284, no. 5, pp. 3443, 2001.
- [2] Berners Lee, Tim *Linked Data*, 2006.
- [3] Lassila, Ora and Swick, Ralph R and others *Resource description framework (RDF) model and syntax specification*, 1998.
- [4] Heath, Tom and Bizer, Christian *Linked data: Evolving the web into a global data space*, Synthesis lectures on the semantic web: theory and technology, vol. 1, no. 1, pp. 1136, 2011.
- [5] Harris, Steve and Seaborne, Andy *SPARQL 1.1 Query Language*, World Wide Web Consortium, 2013.
- [6] Taelman, Ruben and Van Herwegen, Joachim and Vander Sande, Miel and Verborgh, Ruben *Comunica: a modular SPARQL query engine for the web*, in International Semantic Web Conference. Springer, 2018, pp. 239255.
- [7] *Open Geospatial Consortium*, URL: <https://ogc.org>
- [8] *GeoSPARQL support: What is GeoSPARQL*, URL: <http://graphdb.ontotext.com/documentation/standard/geosparql-support.html>
- [9] Shen, Jingwei and Chen, Min and Liu, Xintao *Classification of topological relations between spatial objects in two-dimensional space within the dimensionally extended 9-intersection model*, Transactions in GIS, vol. 22, no. 2, pp. 514541, 2018.

Inhoudsopgave

Lijst van figuren	17
Lijst van tabellen	18
Lijst van listings	19
Lijst van afkortingen	20
1 Inleiding	21
1.1 Overzicht	22
1.2 Probleemstelling en doel	23
1.3 Onderzoeksvraag	23
2 Literatuurstudie	25
2.1 Semantic Web	25
2.2 Linked Data	29
2.2.1 Regels	29
2.2.2 Vijfsterrenmodel	29
2.2.3 Linked data gevisualiseerd	30
2.3 RDF	31
2.3.1 RDF data model	32

2.3.2	RDF serialisatie formaat	33
2.4	SPARQL	40
2.4.1	SPARQL basisvoorbeelden	40
2.4.2	SPARQL functies	41
2.4.3	SPARQL aggregaties	42
2.4.4	SPARQL modifiers	42
2.4.5	SPARQL query forms	42
2.4.6	Conclusie	43
2.5	Comunica	44
2.5.1	Waarom Comunica?	44
2.5.2	Design patterns	45
2.5.3	Architectuur	46
2.5.4	Conclusie	47
2.6	OGC	48
2.6.1	WKT	48
2.6.2	GML	50
2.6.3	GeoSPARQL	51
2.7	GeoSPARQL	52
2.7.1	Vereisten	52
2.7.2	Architectuur	53
2.7.3	Properties	53
2.7.4	Topologische relaties	54
2.7.5	Niet-topologische relaties	57
2.7.6	Query Rewrite Extension	58

<i>INHOUDSOPGAVE</i>	15
3 Implementatie	60
3.1 Communica	60
3.1.1 Sparqlalgebrajs	60
3.1.2 Sparqlee	63
3.1.3 Verbeteringen	63
3.2 Datastructuur	64
3.2.1 GeoJSON	64
3.2.2 Terraformer	64
3.3 Topologische functies	65
3.3.1 Terraformer	65
3.3.2 Manueel	66
3.3.3 Turf.js	70
3.4 Niet-topologische functies	72
3.4.1 Beperkingen	72
3.5 Referentiesysteem	74
3.5.1 Proj4js	74
3.5.2 Beperkingen	74
3.6 Testomgeving	75
3.7 Overzicht	77
3.7.1 Huidige status	77
3.7.2 Toekomstwerk	77
4 Interfaces	79
4.1 Testset	79

4.1.1	Queries	81
4.2	Data dump	84
4.3	Triple pattern fragment interface	85
4.4	SPARQL endpoint	86
5	Conclusie	87
5.1	Toekomstig werk	88
5.2	Tot slot	89
	Bibliografie	90

Lijst van figuren

2.1	Semantic Web Stack (gebaseerd op <i>Semantic Web Stack</i> [3])	26
2.2	Voorbeeld van Linked Data.	31
2.3	Voorbeeld van een RDF statement.	32
2.4	Actor-Mediator-Bus patroon, foto van “Comunica: a Modular SPARQL Query Engine for the Web” [20].	46
2.5	Vereenvoudigd diagram van de GeoSPARQL klassen “Feature” en “Geometry” met sommige properties (figuur gebaseerd op [22]).	53
2.6	Spatiale objecten met hun <i>interior</i> , <i>boundary</i> en <i>exterior</i> : (a) Een punt; (b) Een lijn; (c) Een vlak. Figuur van [23].	54
2.7	Voorbeeld voor de DE-9IM matrix.	55
3.1	Illustratie geospatiale data van [21]	65
3.2	Voorbeeld polygon contains point.	68
3.3	Voorbeeld van polygon contains line.	68
3.4	Voorbeeld van polygon contains polygon.	69
3.5	Problematiek union.	72
3.6	Screenshot van online geplaatste testomgeving.	76
4.1	Testset voor het testen van de verschillende bronnen.	80

Lijst van tabellen

2.1	Primitieve geometrieën.	49
2.2	Meerdelige geometrieën.	50
2.3	Simple Features topologische relaties (tabel van [21]).	57
3.1	Implementatie GeoSPARQL functies (Simple Features familie) met “Turf.js”. . .	71

Lijst van codefragmenten

2.1	Profiel in RDF/XML.	34
2.2	Profiel in RDFa.	35
2.3	Uitgebreidt profiel in Turtle.	37
2.4	Profiel in N-Triples.	38
2.5	Profiel in JSON-LD.	39
2.6	Basis SPARQL query.	40
2.7	SPARQL query die personen vindt met zowel een naam en emailadres hebben. .	41
2.8	Voorbeeld GML bij LineString.	51
2.9	Voorbeeldquery die herschreven zal worden.	59
2.10	Template om queries te herschrijven (codefragment van [21]).	59
3.1	Voorbeeld van GeoSPARQL query.	61
3.2	Voorbeeld van SPARQL algebra.	62
4.1	Query die alle provincies in Vlaanderen zoekt.	81
4.2	Query die geospatiale objecten in Vlaanderen zoekt.	82
4.3	Query die provincies en wegen in België zoekt.	82
4.4	Query die wegen die door Oost-Vlaanderen lopen, zoekt.	82
4.5	Query geospatiale objecten binnen de <i>bounding box</i> van Brabant zoekt.	83

Lijst van afkortingen

CSV *Comma-Seperated Values*. 26

DOM *Document Object Model*. 30

GIS *Geographic Information System*. 47, 48

HTML *HyperText Markup Language*. 30, 31

HTTP *HyperText Transfer Protocol*. 25

TPF *Triple Pattern Fragment*. 20, 40, 43, 81, 83–85

W3C *World Wide Web Consortium*. 23, 26, 29, 30, 36

“The future belongs to those who believe in the beauty of their dreams.”

~Eleanor Roosevelt

1

Inleiding

Het Web zorgt ervoor dat heel wat informatie beschikbaar is voor mensen en gedeeld kan worden door mensen. De webpagina's zijn dan ook makkelijk toegankelijk voor iedereen die beschikt over een internetverbinding. Helaas betekent dit niet dat machines de gegevens even makkelijk kunnen decoderen. Om dagelijkse taken uit te voeren is menselijke interactie dan ook essentieel. Wanneer we zonder tussenkomst van machines een daguitstap plannen dan is dit op zich al een zeer complex proces. Zo moet onder andere de agenda van alle betrokken personen vergeleken worden om te weten te komen wanneer (bijna) iedereen beschikbaar is. Er moet gecontroleerd worden of de weersverwachtingen ideaal zijn voor de uitstap. Ook moet er rekening gehouden worden met de interesses van de verschillende personen om te beslissen welk type uitstap gedaan zal worden. Indien we dit proces willen laten overnemen door *intelligent agents* dan houdt dit per definitie in dat we machines toegang moeten kunnen verlenen tot allerlei persoonlijke informatie.

Verder zijn gegevens ook niet voor iedereen beschikbaar. Grote spelers zoals Google, Instagram, LinkedIn hebben rechtstreeks toegang tot persoonlijke gegevens van mensen. Ze kunnen deze gegevens zelf bijhouden, opvragen en controleren. Wanneer verschillende bedrijven dezelfde gegevens van mensen gaan opslaan, ontstaan er duplicate data. Dit maakt het proces opnieuw complexer. Bovendien moet in deze ook rekening gehouden worden met de wetten op de privacy, die overigens per land verschillend zijn. Een mogelijke oplossing om deze problemen te omzeilen is de verkregen info decentraliseren. Dit wil concreet zeggen dat de gebruiker zelf controle heeft

over zijn gegevens. Bedrijven die info willen over bepaalde personen, zullen dit zelf bij de gebruikers moeten opvragen. De gegevens worden dus niet bijgehouden in echte databanken, maar in andere bronnen waarop verder in deze masterproef op ingegaan zal worden.

Bovendien is het moeilijk om te werken met geografische data, omdat hier weinig implementaties van gemaakt zijn. Het geografische is ook eerder wiskundig om op te lossen. Deze masterproef legt de focus op het werken met geografische gegevens.

1.1 Overzicht

In Hoofdstuk 2 wordt de *state of the art* behandeld, waarbij in details ingegaan wordt op technologieën/technieken al bestaan. Hierbij geeft Sectie 2.1 uitleg over het Semantisch Web. Vervolgens leggen Sectie 2.2, Sectie 2.3 en Sectie 2.4 de gebruikte technologieën uit, namelijk Linked Data, RDF en SPARQL. Sectie 2.5 vertelt meer over Comunica. Comunica brengt de (net hiervoor) genoemde technologieën bij elkaar in een implementatie. Sectie 2.6 vertelt over het OGC. Dit is een organisatie die standaarden voorziet om met geografische data te werken. Ten slotte beschrijft Sectie 2.7 een specifieke standaard van het OGC, namelijk GeoSPARQL. Hierbij is Sectie 2.7 zo belangrijk omdat het onderzoek (zie Sectie 1.3) een implementatie van GeoSPARQL vereist. Zo zal Sectie 2.7 uitleggen waar rekening mee moet gehouden worden om de implementatie te maken.

In Hoofdstuk 3 wordt de eigen implementatie uitgelegd van GeoSPARQL. Zo legt Sectie 3.1 uit waarom Comunica zo een belangrijke rol speelt bij deze implementatie. Dit wordt gevolgd door Sectie 3.2, Sectie 3.3, Sectie 3.4 en Sectie 3.5 die beschrijven welke keuzes gemaakt zijn voor de verschillende aspecten van de implementatie. Daarnaast beschrijft Sectie 3.6 hoe de testomgeving gemaakt is. Ten slotte wordt het voorgaande nogmaals overlopen in Sectie 3.7 om te verduidelijken hoe alles exact samenwerkt. Hierbij wordt ook aangehaald welke verbeteringen nog dienen gemaakt te worden in toekomstig werk.

Vervolgens wordt in Hoofdstuk 4 beschreven hoe het effectieve testen van de onderzoeksvraag gebeurt. Hiervoor wordt de hierboven beschreven testomgeving gebruikt. In Sectie 4.1 wordt uitlegd welke use-case en dataset gebruikt zijn voor het testen van dit onderzoek. Vervolgens wordt in Sectie 4.2, Sectie 4.3 en Sectie 4.4 gecontroleerd of de hypothesen (zie Sectie 1.3) voldaan zijn.

Ten slotte zal in Hoofdstuk 5 een uiteindelijke conclusie getrokken worden. Hier wordt geantwoord op de vraag welke “Linked data publicatie”-interfaces uitgebreid kunnen worden met GeoSPARQL-functionaliteiten door de filtering op de client uit te voeren.

1.2 Probleemstelling en doel

Het creëren van een Semantisch Web staat nog in zijn kinderschoenen, met al enkele jaren onderzoek op de teller. Hoewel er al veel vooruitgang geboekt is, vereist het nog steeds zeer veel werk. Het ophalen van gegevens op het internet is reeds mogelijk door *query engines* zoals onder andere Comunica en Virtuoso. Er is echter een veel beperkter aanbod aan mogelijkheden om met geografische informatie te werken. De bestaande implementaties van GeoSPARQL zijn incompleet of niet voldoende meegaand met de regels die opgesteld zijn door het OGC.

Bovendien is het met huidige implementaties niet mogelijk om te queryen over verschillende bronnen of bronnen van verschillende types. Een simpel voorbeeld om dit probleem te verduidelijken, is het volgende. Stel: de Belgische overheid heeft een dataset die de volledige grens van België beschrijft (aan de hand van OGC standaarden). Daarnaast bevat deze dataset een soortgelijke beschrijving van de gewesten, provincies, gemeenten, steden en wegen. Op die manier zou het mogelijk zijn om op te vragen welke gemeenten of steden binnen een bepaalde provincie liggen. Daarnaast zou het mogelijk zijn om te vragen welke steden of wegen op een bepaalde afstand (of interessanter: een kleinere afstand) van een stad liggen. Veronderstel nu dat de Franse, Nederlandse en Duitse overheden beschikken over een gelijkaardige dataset. Dan zouden soortgelijke opvragingen in deze dataset gedaan kunnen worden. Het zou echter onmogelijk zijn te weten welke Franse (of Nederlandse of Duitse) steden op een bepaalde afstand van een Belgische stad liggen.

Dit probleem zou niet voorkomen wanneer de techniek van Comunica uitgebreid kan worden naar de functionaliteiten van GeoSPARQL. Deze masterproef zal een simpele dataset voorzien die een soortgelijk probleem als hierboven kan simuleren. Hierbij zal gebruik gemaakt worden van Comunica, om zo gebruik te maken van de reeds voorziene mogelijkheden om te queryen over heterogene interfaces.

1.3 Onderzoeksvraag

Zoals beschreven in Sectie 1.2 zal onderzocht worden in welke mate Comunica uitgebreid kan worden om de GeoSPARQL functionaliteiten te ondersteunen. Aangezien meerdere bronnen van verschillende types gebruikt kunnen worden, moet de filtering zelf op de client gebeuren. Dit wordt geformuleerd in een onderzoeksvraag die uiteindelijk in Hoofdstuk 5 beantwoord zal worden.

Onderzoeksvraag Welke “Linked Data publicatie”-interfaces kunnen uitgebreid worden met GeoSPARQL-functionaliteiten door de filtering op de client uit te voeren?

Wanneer Comunica zelfstandig een bestand moet ophalen en vervolgens queryen, is het mogelijk dat dit bestand geografische gegevens bevat. Deze queries moeten afgehandeld kunnen worden op zo een manier dat topologische (en niet-topologische) relaties berekend kunnen worden. Deze bron bevat zelf geen logica en wordt vervolgens de *baseline*.

Hypothese 1 Het is mogelijk om GeoSPARQL queries uit te voeren over “data dumps” waarbij de filtering op de client-side gebeurt.

Wanneer de bron een *Triple Pattern Fragment* (TPF) interface is, is er een server die de gegevens aanbiedt. Hierbij is het opnieuw mogelijk dat de dataset geografische informatie bevat. Door het filteren op de server moet het wederom mogelijk zijn om GeoSPARQL opvragingen uit te voeren.

Hypothese 2 Het is mogelijk om GeoSPARQL queries uit te voeren over “TPF interfaces” door de filtering op de client uit te voeren.

Een dataset kan ook vrijgegeven worden aan de hand van een SPARQL *endpoint*. Comunica heeft de functionaliteit om te queryen naar een SPARQL *endpoint*. Hierbij is het niet vanzelfsprekend dat GeoSPARQL queries opgevraagd kunnen worden aan een SPARQL *endpoint*. Dit is echter wel mogelijk door de filtering op de client-side te doen.

Hypothese 3 Het uitvoeren van GeoSPARQL queries op een “SPARQL *endpoint*” is niet vanzelfsprekend. Het is echter mogelijk door de filtering op de client uit te voeren.

“Knowledge is power.”

~Francis Bacon

2

Literatuurstudie

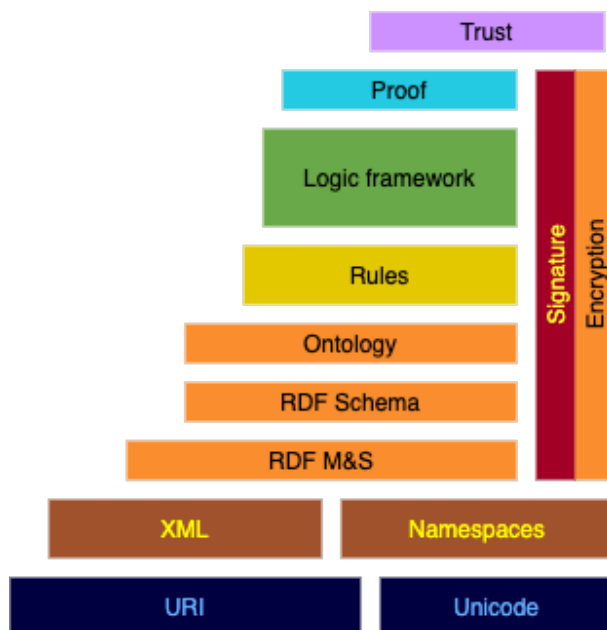
2.1 Semantic Web

In 2001 spreekt Tim Berners-Lee (uitvinder van het World Wide Web) over een nieuwe revolutie. Dit is de eerste introductie van het *Semantic Web* (soms wordt er ook naar verwezen onder de term *Web 3.0*). Hierbij zou het web, zoals het toen was, evolueren. Zo is het web altijd leesbaar geweest voor mensen, maar niet interpreteerbaar door machines. Het *Semantic Web* zou hier verandering in brengen. Zo zouden machines het web, net zoals mensen, kunnen interpreteren. Deze machines heten *intelligent agents* en zij moeten in staat zijn om complexe taken volledig autonoom uit te voeren [1].

Om dit mogelijk te maken zijn er verschillende stappen nodig. Als eerste moet ervoor gezorgd worden dat er betekenis gegeven kan worden op een manier die computers kunnen begrijpen. Ook moet deze kennis representeerbaar zijn voor machines. Hiervoor wordt gebruik gemaakt van het RDF model (zie Sectie 2.3) met behulp van bijvoorbeeld XML. Verder is het ook belangrijk om er rekening mee te houden dat informatie uit verschillende databanken een andere terminologie gebruikt om hetzelfde uit te drukken. Hiervoor worden verschillende ontologieën gehanteerd (een definitie van de term ontologie wordt gegeven in Subsubsectie 2.1). De kracht van het *Semantic Web* zal zichtbaar zijn wanneer er programma's gemaakt worden die informatie kunnen verzamelen van verschillende bronnen (de zogenaamde *intelligent agent*) [1].

Een belangrijk aspect om het *Semantic Web* mogelijk te maken is dus decentralisatie. Hiermee wordt bedoeld dat de macht (hier in de vorm van informatie) niet in handen mag zijn van enkele grote spelers, maar verspreid moet worden. In een ideale vorm van het *Semantic Web* zou elke persoon een *pod* hebben die de informatie over zichzelf bevat. Wanneer een website toegang tot deze informatie zou willen, dan zou deze informatie uit de *pod* opgehaald moeten worden. Dit zou nog andere voordelen bieden, waaronder een verbeterde privacy (toegang verlenen aan wie de persoon wil).

De architectuur van het *Semantic Web* gebaseerd is op een hiërarchie van talen, waarbij elke taal de mogelijkheden van de talen lager in deze hiërarchie optimaal zal benutten en uitbreiden. Deze hiërarchie is gevisualiseerd in Figuur 2.1, ontworpen door Tim Berners-Lee. In de paper “Semantic Web Architecture: Stack or Two Towers?” worden alternatieve voorstellingen van de *Semantic Web Stack* besproken [2]. In deze masterproef wordt niet verder ingegaan op deze uitbreidingen. De lagen van de oorspronkelijke *Semantic Web Stack* die belangrijk zijn voor deze masterproef, worden hieronder besproken.



Figuur 2.1: Semantic Web Stack (gebaseerd op *Semantic Web Stack* [3])

In wat volgt zullen de talen die van belang zijn voor deze masterproef bespreken.

Unicode

Unicode is een systeem dat gebruikt wordt voor het encoderen van karakters. Net zoals ASCII is het ontwikkeld om ontwikkelaars te ondersteunen bij het maken van applicaties. Unicode zorgt voor de codering van karakters en pakt hierbij de problemen aan van eerdere karakter encodeer

systemen, zoals onder meer het niet ondersteunen van alle karakters. Zo zal unicode een uniek nummer hebben voor elk karakter op elk platform, voor elk programma en in elke taal [4].

Unicode ligt aan de basis van de *Semantic Web Stack* omdat het *Semantic Web* documenten in verschillende talen moet kunnen doorgeven. Deze documenten moeten dus ook kunnen gerepresenteerd worden.

URI

URI staat voor *Uniform Resource Identifier*. Dit is een uniforme manier voor het identificeren van objecten. Deze term wordt soms door elkaar gehaald met de term URL, wat staat voor *Uniform Resource Locator*. Het grote verschil tussen beiden is dat een URI een object kan identificeren (= hoe iets te benoemen), terwijl een URL een object kan localiseren (= waar iets te vinden). De verwarring tussen beiden komt door hun onderlinge relatie. Om dit verschil te begrijpen is het belangrijk te weten dat de verzameling van URL's een subset is van alle URI's. Zo is elke URL een URI, maar niet omgekeerd [5].

Samen met unicode ligt URI mede aan de basis van de *Semantic Web Stack*. Zowel URI als unicode maken het mogelijk om op het Web resources te identificeren op eenzelfde eenvoudige manier.

XML

XML staat voor *Extensible Markup Language*. Het wordt gebruikt voor de beschrijving van data. Eén van de belangrijkste kenmerken van de XML standaard is het vermogen om op een zeer flexibele manier data te structureren. Het *World Wide Web Consortium* (W3C) beveelt XML dan ook aan. XML werkt aan de hand van elementen die gedefinieerd worden door *tags*. Zo heeft elk element een begin- en een eind*tag*. XML ondersteunt ook geneste elementen zodat echte hiërarchiën gemaakt kunnen worden. XML is dus belangrijk gezien zijn eenvoud en uitbreidbaarheid [6].

Namespaces

XML Namespaces worden ook aanbevolen door het W3C. De reden hiervoor is om te voorkomen dat verschillende elementen dezelfde en dus conflicterende namen hebben. Op deze manier wordt de woordenschat gedifferentieerd, zodat deze woordenschat herbruikt kan worden. Het idee van namespaces steunt volledig op de werking van URI [7].

RDF Model, Syntax en Schema

RDF staat voor *Resource Description Framework*. RDF zal op een beschrijvende manier informatie geven. RDF is echter te belangrijk om kort besproken te worden en zal dus uitvoerig besproken worden in Sectie 2.3.

Ontology

Het woord *ontology* zorgt voor veel verwarring en heeft bijgevolg al meerdere verschillende definities gekregen. In zijn artikel “What is an ontology?” beschrijft Tom Gruber een ontologie als een specificatie van een conceptualisatie. De term ontologie komt van de filosofie waar het de betekenis heeft van een systematisch teken van het bestaan. Een ontologie kan beschreven worden als het definiëren van een set van representerende termen. Zo zullen relaties tussen objecten beschreven worden in een vorm die begrijpbaar is door mensen. Formeel betekent dit dat een ontologie een uitspraak is van een logische theorie [8].

In de computerwetenschappen refereert de term ontologie naar een formele beschrijving van kennis. Zo kan informatie die van verschillende bronnen komt vertrouwen op de ontologieën om een gelijkaardige betekenis te krijgen.

2.2 Linked Data

Het semantisch web gaat echter niet enkel over het plaatsen van data op het web. Het belangrijkste aspect van het semantisch web is het maken van links, zodat zowel personen als machines het web van data kunnen doorkruisen. Het belangrijke aan gelinkte data kan als volgt omschreven worden: wanneer je data hebt, kan je er elders andere gerelateerde data mee vinden. Op deze manier wordt context aan de data meegegeven. Bij gelinkte data worden deze links beschreven aan de hand van RDF. Hierbij worden URI's gebruikt voor het identificeren van objecten. Om data te interconnecteren zijn er vier regels, met als doel dat de informatie in de toekomst op onvoorspelbare manieren herbruikt zou kunnen worden. Daarnaast is het belangrijk dat de data open en toegankelijk zijn om herbruikt te worden [9].

2.2.1 Regels

Bij gelinkte data zijn er vier regels waaraan voldaan moet worden:

1. Objecten moeten geïdentificeerd worden met URI's. Dit is nodig om te kunnen spreken over een semantisch web [9].
2. Er moet gebruik gemaakt worden van *HyperText Transfer Protocol* (HTTP) URI's. Dit is nodig zodat andere gebruikers de namen zouden kunnen opzoeken [9].
3. Bijhorende informatie moet gevonden kunnen worden wanneer een URI gevolgd wordt. Dit is in het basisformaat van RDF en XML. Deze kan ook doorzocht worden aan de hand van SPARQL (verder besproken in Sectie 2.4), dit is een query service voor gelinkte data in RDF formaat [9].
4. Er moeten links voorzien worden naar andere locaties die gelijkaardige data bevatten, zodat deze opgezocht kunnen worden. Deze laatste regel is belangrijk om de informatie op het web te connecteren [9].

2.2.2 Vijfsterrenmodel

Het vijfsterrenmodel is een manier om informatie in te delen op basis van openheid. Meer sterren betekent dat de informatie meer open is. Tim Berners-Lee stelde dit model voor als schema voor gelinkte open data. Gelinkte open data is een essentieel onderdeel van het semantisch web.

Eén ster stelt hetvolgende: “*Available on the web but with an open licence, to be Open Data*”. Dit betekent dat gebruikers informatie kunnen ophalen, gebruiken en delen met iedereen. Het

gaat hier echter louter over het delen van informatie, het maakt dus niet uit in welk formaat dit komt [9].

Twee sterren stelt dan weer: “*Available as machine-readable structured data*”. Om twee sterren te krijgen is het belangrijk dat de informatie een bepaalde structuur heeft, zodat machines deze informatie kunnen verwerken. Dit kan bijvoorbeeld zijn in de vorm van een excel spreadsheet. Dit soort informatie is echter nog steeds vrij gesloten aangezien de gebruikers afhankelijk zijn van bepaalde software om toegang te krijgen tot de informatie [9].

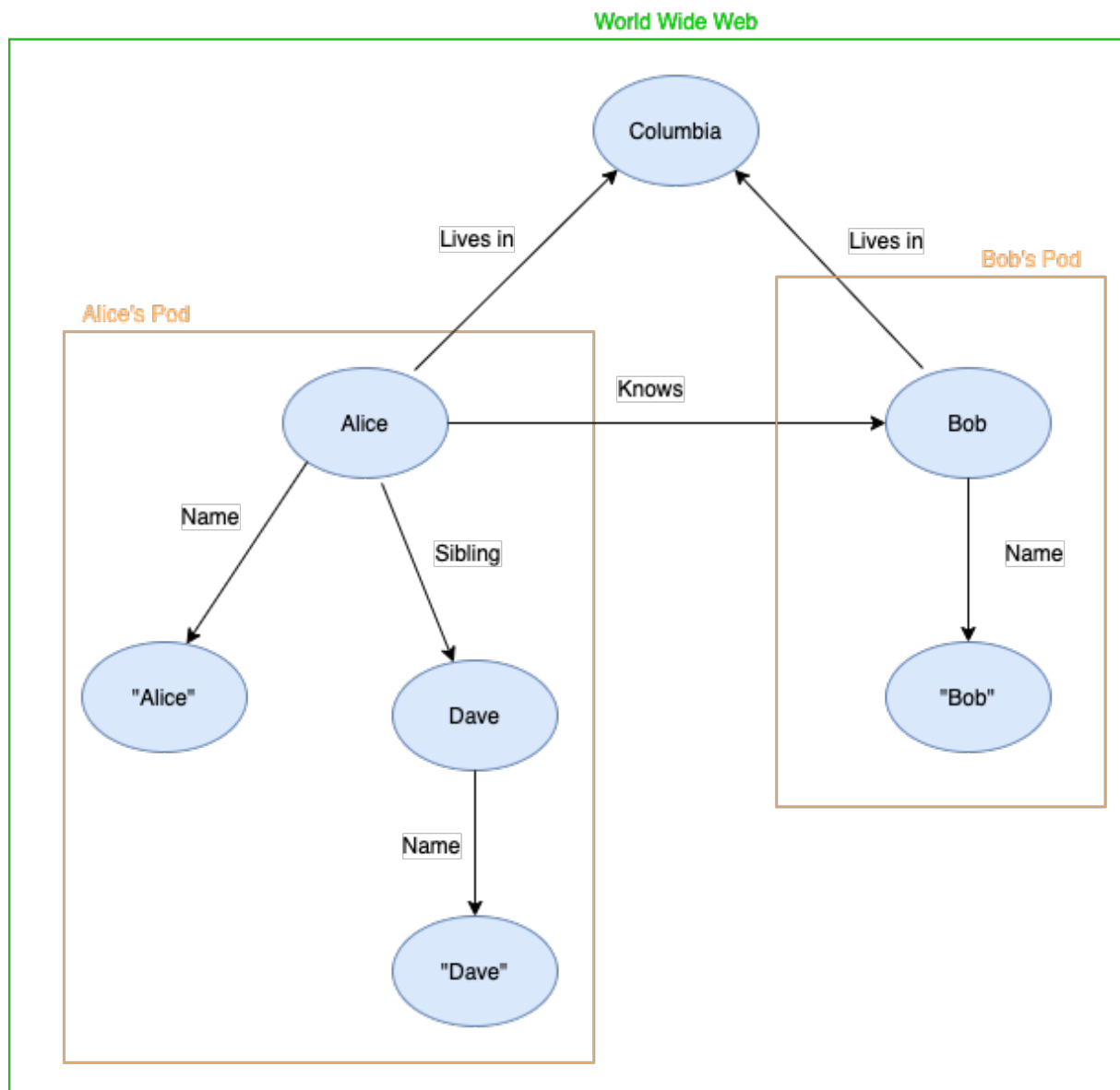
Drie sterren betekent: “*The same as 2 stars, plus non-proprietary format*”. Het verschil om van twee sterren naar drie sterren te stijgen is het vermijden van de nood aan specifieke software om de informatie te bemachtigen. Dit kan bijvoorbeeld door de informatie op te slaan in *Comma-Separated Values* (CSV) formaat [9].

Vier sterren is vervolgens: “*All the above plus, use open standards from W3C to identify things, so that people can point at your stuff*”. Om een vierde ster te verdienen moet de informatie voldoen aan de open standaarden van W3C. Zo moet het objecten identificeren aan de hand van RDF of SPARQL. Hierbij is het belangrijk dat gebruikers (aan de hand van URI) kunnen verwijzen naar de data [9].

Tenslotte betekent vijf sterren het volgende: “*All the above, plus: Link your data to other people’s data to provide context*”. Om de laatste ster ook te kunnen behalen, dient men de informatie te linken aan bijhorende informatie in een andere context. Op deze manier worden de links verder verspreid. Hier wordt er dus letterlijk verwezen naar andere locaties, met als doel om meer context terug te vinden [9].

2.2.3 Linked data gevisualiseerd

Een voorbeeld van hoe het *World Wide Web* eruit zou kunnen zien is geschetst in Figuur 2.2. Dit voorbeeld toont de ideale situatie waar personen een eigen pod met informatie hebben. Zo hebben Alice en Bob elk hun eigen plaats in het web, waar informatie over hun te vinden is. Deze informatie zou onder andere hun naam, telefoonnummer, adres, interesses, werkomgeving, etc kunnen zijn. Daarnaast zijn er ook connecties tussen Alice en Bob. Om te beginnen is Bob gekend door Alice, waardoor er een verwijzing is naar meer context over Bob in zijn pod. Daarnaast wonen ze beide in dezelfde stad, waardoor het mogelijk is om bijvoorbeeld te zoeken naar iedereen die in een bepaalde stad woont. Al deze personen zullen terug te vinden zijn aan de hand van een verwijzing naar meer context (lees: meer informatie) gelinkt aan personen. Dit is echter een vereenvoudigd voorbeeld, in de reële situatie zijn er veel meer links en dit in meerdere richtingen.



Figuur 2.2: Voorbeeld van Linked Data.

2.3 RDF

RDF staat voor *Resource Description Framework*. Het *World Wide Web* is gemaakt voor mensen, en hoewel machines het kunnen lezen kunnen ze het niet altijd interpreteren. Het doel van RDF is om een algemene methode te voorzien om relaties tussen data objecten te beschrijven. Zo is RDF ontstaan in een poging om metadata te maken. Metadata worden gezien als data over data, maar kunnen beter geïnterpreteerd worden als data die *web resources* beschrijven. RDF blijkt een zeer effectieve manier om informatie van verschillende bronnen te kunnen integreren door de

informatie los te koppelen van zijn schema. Op deze manier kunnen de gegevens ook tegelijkertijd opgezocht worden. Zo poogt het dus om informatie op het web interpreteerbaar te maken voor machines. RDF steunt op de bestaande web standaarden zoals XML en URI. XML is echter slechts een mogelijke syntax. Er bestaan verschillende andere manieren mogelijk om dezelfde RDF data te representeren. Het algemeen doel van RDF is het definiëren van een mechanisme. Dit mechanisme zorgt voor het beschrijven van *resources* die geen veronderstellingen maken van een specifiek domein, noch een semantiek definiëren [10].

2.3.1 RDF data model

De onderliggende structuur van een RDF uitdrukking is een collectie van triples. Elk van deze triples bestaat uit een *subject* (= onderwerp), *predicate* (= eigenschap) en *object* (= voorwerp). Zoals te zien is in Figuur 2.3, kan dit geïllustreerd worden als een *node-arc-node link* (*node* is een knoop, terwijl *arc* een tak is). Deze collectie van triples kan bijgevolg gezien worden als een graaf. Hierbij is de richting van de *arc* belangrijk, deze wijst in de richting van het *object*. [11].

Eén enkel triple weerspiegelt een eenvoudige zin. Bij een kort terugblikken naar Figuur 2.2 is het volgende triple te zien: (*Alice* - *Knows* - *Bob*). Deze triple staat letterlijk voor de zin “*Alice knows Bob*”. Deze zin hoeft echter niet altijd een exacte vertaling te zijn, bij een ander voorbeeld is te zien dat er enkele korte woorden toegevoegd moeten worden om een gramaticaal correcte zin te bekomen: (*Alice* - *Name* - “*Alice*”) wordt dan weer “*Alice has name Alice*”. In dit voorbeeld gaat het nu over de naam, maar het kan hier evenwel over een emailadres of een leeftijd gaan.

URI-gebaseerde vocabulair

Een *node* kan een URI, een *literal* of *blank* zijn. Een URI referentie of een *literal* die gebruikt wordt als *node* identificeert waar de *node* voor staat. Een URI referentie die gebruikt wordt als *predicate* beschrijft dan weer de relatie tussen de “dingen” in de *nodes* die geconnecteerd worden. Een *blank node* is een *node*, die louter staat voor een unieke code die gebruikt kan worden in één of meer RDF uitdrukkingen [11].



Figuur 2.3: Voorbeeld van een RDF statement.

Literals

Literals worden gebruikt om waarden zoals nummers en datums te identificeren. Elke *literal* kan echter ook voorgesteld worden door een URI, maar vaak is het intuïtiever om een *literal* te gebruiken. Een *literal* kan enkel in het *object* van de RDF uitdrukking staan, dus niet in het *subject* of *predicate*. Er bestaan twee soorten *literals* [11]:

1. **Plain literal:** deze *literal* staat voor een *string* die gecombineerd is met een optionele taal *tag*. Deze wordt gebruikt om gewone tekst weer te geven en eventueel bij te plaatsen in welke taal deze tekst is.
2. **Typed literal:** deze *literal* staat voor een *string* die gecombineerd is met een *datatype* URI. Deze URI wordt gebruikt om aan te duiden hoe deze informatie geïnterpreteerd moet worden.

Twee literals zijn gelijk indien alle volgende regels voldoen [11]:

- Beide strings zijn identiek;
- Ofwel hebben beide of geen van beide taal tags;
- Als ze taal tags hebben moeten deze identiek zijn;
- Ofwel hebben beide of geen van beide *datatype* URIs;
- Als ze *datatype* URIs hebben moeten deze identiek zijn.

2.3.2 RDF serialisatie formaat

Het is belangrijk te onthouden dat RDF geen dataformaat is, maar een datamodel. Het is een beschrijving dat de gegevens zich moeten voorstellen in de vorm van (*subject*, *predicate*, *object*) triples. Alvorens men een RDF-graaf kan publiceren, zullen de data geserialiseerd moeten worden, gebruik makend van een RDF-syntax. De W3C heeft verschillende formaten gestandaardiseerd, welke hieronder vermeld zijn. Er zijn welliswaar nog meer mogelijkheden. Bij elk van deze mogelijkheden zal hetzelfde voorbeeld telkens herschreven worden in een ander formaat [12].

Om de verschillende formaten duidelijk te maken (gebruik makend van principes zoals URIs, *literals* zowel met als zonder *datatype*) zal er bij de verschillende formaten éénzelfde voorbeeld uitgeschreven staan. Dit voorbeeld gaat over hoe het profiel van een persoon eruit zou kunnen zien. Aangezien het Turtle formaat het meest leesbare is, is enkel hier de uitgebreide versie

zichtbaar. Bij andere formaten is dit profiel sterk ingekort. Bij dit voorbeeld zijn ook verschillende ontologieën gebruikt, zoals onder andere “foaf” en “dbo”. Bovendien is bij dit voorbeeld ook te zien dat de *predicate* “a” gebruikt wordt. Dit is een alternatief voor “rdf:type”, maar verder volledig equivalent.

RDF/XML

De RDF/XML syntax is gestandaardiseerd door het W3C en wordt wijd gebruikt om Linked Data te publiceren op het web. Deze syntax wordt echter gezien als moeilijk te lezen en te schrijven voor mensen, waardoor deze steeds minder vaak gebruikt wordt. Bij deze syntax wordt het RDF-datamodel voorgesteld aan de hand van XML [13].

Een ingekorte versie van het hierboven vermelde voorbeeld in RDF/XML-formaat is te zien in Codefragment 2.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:dbo="http://dbpedia.org/ontology/"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
  <rdf:Description rdf:about="https://example.org/profile/dreeki/#me">
    <foaf:familyName xml:lang="nl">De Witte</foaf:familyName>
    <dbo:birthDate rdf:datatype="http://www.w3.org/2001/XMLSchema#date">1994-08-27</dbo:birt
    <foaf:familyName xml:lang="en">De Witte</foaf:familyName>
    <foaf:givenName xml:lang="en">Andreas</foaf:givenName>
    <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
    <foaf:givenName xml:lang="nl">Andreas</foaf:givenName>
  </rdf:Description>
</rdf:RDF>
```

Codefragment 2.1: Profiel in RDF/XML.

RDFa

RDFa is dan weer een serialisatieformaat dat de RDF triples zal integreren in *HyperText Markup Language* (HTML) documenten. In eerdere pogingen om RDF en HTML te mixen werden de RDF triples geïntegreerd in de *comments*. Dit is hierbij niet het geval. Bij RDFa zijn de RDF triples verweven in de *HTML Document Object Model* (DOM). Dit betekent dat de bestaande

inhoud van de pagina's aangeduid wordt met RDFa door de HTML code aan te passen. Hierdoor worden de gestructureerde data blootgesteld aan het web [14].

Een ingekorte versie van het hierboven vermelde voorbeeld in RDFa formaat is te zien in Codefragment 2.2.

```
<div xmlns="http://www.w3.org/1999/xhtml"
  prefix="
    foaf: http://xmlns.com/foaf/0.1/
    rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
    dbo: http://dbpedia.org/ontology/
    xsd: http://www.w3.org/2001/XMLSchema#
    rdfs: http://www.w3.org/2000/01/rdf-schema#"
>
<div typeof="foaf:Person" about="https://example.org/profile/dreeki/#me">
  <div property="foaf:familyName" xml:lang="nl" content="De Witte"></div>
  <div property="foaf:familyName" xml:lang="en" content="De Witte"></div>
  <div property="foaf:givenName" xml:lang="nl" content="Andreas"></div>
  <div property="foaf:givenName" xml:lang="en" content="Andreas"></div>
  <div property="dbo:birthDate" datatype="xsd:date" content="1994-08-27"></div>
</div>
</div>
```

Codefragment 2.2: Profiel in RDFa.

Turtle

Turtle is een *plain text* formaat voor de serialisatie van RDF-gegevens. Turtle voorziet prefixen voor *namespaces* en andere verkortingen. Zo worden de prefixen bovenaan geschreven en moet elk triple eindigen op een “:”, “;” of “.”. Een “:” betekent dat het volgende triple volledig los staat van het huidige triple. Een “;” betekent dat het volgende triple hetzelfde *subject* heeft als het huidige triple, waardoor er slechts twee waarden (*predicate* en *object*) op de volgende lijn staan. tenslotte betekent een “.” dat het volgende triple hetzelfde *subject* en *predicate* heeft als het huidige triple, waardoor er slechts één waarde (*object*) op de volgende lijn staat. Deze verkortingen zijn echter geen verplichting. Aangezien Turtle zowel zeer leesbaar als schrijfbaar is, wordt deze in de meeste visuele teksten gebruikt. Vanwege de leesbaarheid zal dit formaat in de rest van deze masterproef (op de bijlagen na) ook gebruikt worden [15].

Een uitgebreide versie van het hierboven vermelde voorbeeld in Turtle formaat is te zien in Codefragment 2.3.

N-Triples

Het N-Triples-formaat is een subset van Turtle. Hierbij zijn de *features* zoals prefixen en verkortingen weggelaten. Het valt het op dat dit serialisatie formaat veel redundantie heeft, zoals alle URIs die in elk triple volledig moeten worden gespecificeerd. Hierdoor zijn deze N-Triples-bestanden veel groter dan overeenkomende Turtle-bestanden. Naast het nadeel van grotere bestanden heeft deze redundantie ook een zeer groot voordeel. Dankzij de redundantie is het mogelijk om N-Triples-bestanden lijn per lijn te overlopen, waardoor het ideaal is om bestanden die te groot zijn om volledig in het geheugen te laden te verwerken. Daarnaast zijn N-Triples ook zeer ontvankelijk voor compressie, waardoor het netwerkverkeer gereduceerd wordt bij het uitwisselen van bestanden. Het N-Triples-formaat is zo de standaard om zeer grote dumps van Linked Data uit te wisselen (bijvoorbeeld voor backup doelen) [16].

Een ingekorte versie van het hierboven vermelde voorbeeld in N-Triples-formaat is te zien in Codefragment 2.4. Hierbij zijn de lijnen gesplitst zodat deze op het blad zouden passen.

JSON-LD

JSON-LD staat voor JSON-LinkedData en is een *lightweight* Linked Data formaat. JSON-LD is makkelijk leesbaar en schrijfbaar. Het is gebaseerd op het al langer bestaande JSON-formaat. Aangezien JSON al langer gebruikt wordt om data door te geven, is JSON-LD het ideale formaat om Linked Data uit te wisselen in een programmeeromgeving. Aangezien het dezelfde syntax heeft als JSON, kan het zonder andere software te installeren onmiddellijk gebruikt worden om RDF data te parsen en te manipuleren. Omdat JSON-LD zo handig in gebruik is, zal dit het meest gebruikte formaat zijn bij de implementaties die gemaakt zijn bij deze masterproef [17].

Een (nog sterker) ingekorte versie van het hierboven vermelde voorbeeld in JSON-LD-formaat is te zien in Codefragment 2.5.

```

@prefix : <https://example.org/profile/dreeki/#>.
@prefix p: <https://data.example.org/people/>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix dbp: <http://dbpedia.org/resource/>.
@prefix dbo: <http://dbpedia.org/ontology/>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

<https://example.org/profile/dreeki/> a foaf:Document,
    foaf:PersonalProfileDocument;
    rdfs:label "Andreas De Witte's FOAF profile"@en;
    foaf:maker :me;
    foaf:primaryTopic :me.

:me a foaf:Person;
    foaf:name "Andreas De Witte"@en, "Andreas De Witte"@nl;
    rdfs:label "Andreas De Witte"@en, "Andreas De Witte"@nl;
    foaf:nick "dreeki"@en, "dreeki"@nl;
    foaf:givenName "Andreas"@en, "Andreas"@nl;
    foaf:familyName "De Witte"@en, "De Witte"@nl;
    dbo:birthPlace dbp:Aalst;
    dbo:birthDate "1994-08-27"^^xsd:date.

p:demian_dekoninck a foaf:Person;
    foaf:givenName "Demian"@en;
    foaf:familyName "Dekoninck"@en;
    rdfs:label "Demian Dekoninck"@en.
:me foaf:knows p:demian_dekoninck.

```

Codefragment 2.3: Uitgebreidt profiel in Turtle.

```

<https://example.org/profile/dreeki/#me>
  <http://xmlns.com/foaf/0.1/givenName> "Andreas"@nl .

<https://example.org/profile/dreeki/#me>
  <http://xmlns.com/foaf/0.1/givenName> "Andreas"@en .

<https://example.org/profile/dreeki/#me>
  <http://dbpedia.org/ontology/birthDate>
    "1994-08-27"^^<http://www.w3.org/2001/XMLSchema#date> .

<https://example.org/profile/dreeki/#me>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Person> .

<https://example.org/profile/dreeki/#me>
  <http://xmlns.com/foaf/0.1/familyName> "De Witte"@nl .

<https://example.org/profile/dreeki/#me>
  <http://xmlns.com/foaf/0.1/familyName> "De Witte"@en .

```

Codefragment 2.4: Profiel in N-Triples.

```
{
  "@context": {
    "dbo": "http://dbpedia.org/ontology/",
    "foaf": "http://xmlns.com/foaf/0.1/",
    "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
    "xsd": "http://www.w3.org/2001/XMLSchema#"
  },
  "@id": "https://example.org/profile/dreeki/#me",
  "@type": "foaf:Person",
  "dbo:birthDate": {
    "@type": "xsd:date",
    "@value": "1994-08-27"
  },
  "foaf:givenName": {
    "@language": "nl",
    "@value": "Andreas"
  }
}
```

Codefragment 2.5: Profiel in JSON-LD.

2.4 SPARQL

SPARQL komt oorspronkelijk van “Simple Protocol And RDF Query Language”, maar aangezien het te uitgebreid werd om nog “*simple*” te noemen, is het veranderd naar het recursieve acroniem “SPARQL Protocol And RDF Query Language” [18]. Dit betekent letterlijk dat SPARQL een zoektaal is voor de opzoeking van RDF gebaseerde gegevens. Hierdoor is het ook zeer makkelijk om meerdere bronnen met RDF-gegevens te combineren. Om het gebruiksgemak te verhogen is SPARQL zeer gelijkaardig aan het meer bekende SQL. Om de vergelijking met SQL even door te trekken, kan RDF-data beschouwd worden als een soort tabel met drie kolommen: de *subject*-kolom, de *predicate*-kolom en de *object*-kolom. Hierbij zou het *subject* analoog zijn aan een entiteit bij SQL. Het *predicate* zou dan opnieuw staan voor welk veld (dus de kolom in de SQL tabel) een waarde heeft en het *object* zou de effectieve waarde van dat veld zijn [19].

Er zijn echter ook belangrijke verschillen met SQL. Zo is de waarde van het *object* vaak geïmpliceerd door de *predicate* waarde. Daarnaast kunnen er ook meerdere (verschillende) *object* waarden zijn voor hetzelfde *predicate*, om zo een lijst te bekomen [19].

Bovendien is SPARQL de W3C aanbeveling als RDF query taal. De traditionele manier om een SPARQL *query processor* te implementeren is door deze te gebruiken als interface voor een onderliggende databank. Dit wordt een SPARQL *endpoint* genoemd. Dit is ook weer te vergelijken met hoe een SQL interface toegang geeft tot een relationele databank.

2.4.1 SPARQL basisvoorbeelden

Aangezien SPARQL gebruikt wordt om RDF gebaseerde gegevens op te vragen, zullen de *statements* in de query zelf ook een RDF vorm hebben. Deze queries zijn syntactisch zeer gelijkend op het Turtle formaat. Zo is de simpelste vorm van een SPARQL query te zien in Codefragment 2.6. Deze zal letterlijk alle triples opvragen en weergeven.

```
SELECT *  
WHERE { ?s ?p ?o. }
```

Codefragment 2.6: Basis SPARQL query.

Er zijn drie variabelen (namelijk “?s”, “?p” en “?o”), die achtereenvolgens weergegeven worden. Elk van deze variabelen kan ook aangepast worden naar een literal of een URI, om zo de query meer zinvol te maken. Zo is Codefragment 2.7 een meer zinvol voorbeeld. Hierin zal gekeken worden naar alle personen in de dataset (via foaf:Person). Vervolgens zal de naam en het emailadres opgehaald worden en in een variabele geplaatst worden, om zo ten slotte deze naam en email te laten zien. Alles bij elkaar zal Codefragment 2.7 van alle personen in de dataset

de naam en het emailadres laten zien.

```
1      PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2      SELECT ?name ?email
3      WHERE {
4          ?person a foaf:Person.
5          ?person foaf:name ?name.
6          ?person foaf:mbox ?email.
7      }
```

Codefragment 2.7: SPARQL query die personen vindt met zowel een naam en emailadres hebben.

Het voorbeeld uit Codefragment 2.7 kan uiteraard nog ingekort worden, zo kan “?person” nog verwijderd worden op lijn 5 en lijn 6, door op lijn 4 en lijn 5 de “.” te veranderen door een “;”. Daarnaast heeft SPARQL ook *blank nodes* (voorgesteld door “[]”). Deze doen zich voor als variabelen, maar zijn het eigenlijk niet. Op deze manier kan zelfs de “?person” van lijn 4 weggelaten worden, door deze lijn tussen vierkante haken te plaatsen. Dit is mogelijk omdat de inhoud van “?person” niet getoond wordt, dus is deze niet echt nodig. Bij deze laatste vorm kan er bij de “SELECT” statement ook een “*” geplaatst worden, omdat er maar twee variabelen meer zijn. Zo krijgen we toch nog steeds de verwachte uitkomst van de query. Dit komt echter de leesbaarheid van het geheel niet ten goede, waardoor er meestal toch gekozen wordt voor de lange versie [19].

2.4.2 SPARQL functies

Functies

Verder ondersteunt SPARQL verschillende functies, waaronder filterfuncties, om zo onder andere verder onderscheid te maken tussen welke lijnen al dan niet tot het resultaat mogen behoren. Deze functies kunnen bijvoorbeeld de waarde van een variabele veranderen of twee variabelen samenvoegen tot een nieuwe variabele. De filterfuncties kunnen dan bijvoorbeeld kijken naar de waarde of deze overeenkomt met een regex. Het werk vereist om uitbreidingen op SPARQL aan te brengen zal voornamelijk bestaan uit het definiëren van nieuwe functies [19].

Matching alternatieven

Naast filterfuncties bestaan er ook verschillende alternatieven, zoals het gebruik van onder andere “UNION” (zeker niet te verwarren met de union functie van GeoSPARQL, besproken in Sectie 2.7) [19].

Negaties

In SPARQL is het zelfs mogelijk om negaties te gebruiken (wat nog steeds lijkt op SQL), zoals een “FILTER NOT EXISTS” of “MINUS”. Er is echter wel een zeer groot verschil tussen deze twee opties. Hierbij zal “FILTER NOT EXISTS” kijken of de waarden gelijk zijn, zodat deze verwijderd kunnen worden. De “MINUS” optie zal dan weer kijken of er een *binding* (= gelijke variabele) aanwezig is tussen de twee gescheiden delen. Indien niet zal “MINUS” niets verwijderen [19].

2.4.3 SPARQL aggregaties

Net zoals SQL heeft SPARQL ook de mogelijkheid om meerdere rijen te aggregeren. Hiervoor wordt gebruik gemaakt van de syntax “GROUP BY”. De mogelijke aggregaatsfuncties zijn dan onder andere “COUNT”, “SUM”, “MIN”, “MAX” en “AVG” [19].

2.4.4 SPARQL modifiers

SPARQL voorziet nog vele functionaliteiten. Enkele voorbeelden zijn [19]:

- “ORDER BY” voor het ordenen van de uitkomst;
- “DISTINCT” om unieke resultaten te bekomen;
- “LIMIT” om aan te geven hoeveel van de eerste rijen teruggegeven mogen worden.

2.4.5 SPARQL query forms

SPARQL heeft vier verschillende query vormen. Deze vormen zullen beslissen hoe het resultaat van een query eruit ziet. Deze vormen zijn [19]:

- “SELECT” wordt gebruikt om alle of een deel van de variabelen uit de query weer te geven;
- “CONSTRUCT” dient dan weer om een geldige RDF-graaf te construeren;
- “ASK” is de meest eenvoudige vorm en wordt gebruikt om te controleren of er een resultaat is voor een bepaalde query. Deze geeft dus een *boolean* waarde terug;
- “DESCRIBE” geeft een RDF-graaf terug die de gevonden *resources* beschrijft .

2.4.6 Conclusie

SPARQL is een query-taal die gebruikt wordt om gegevens op te halen van het Web. Om een correcte implementatie van SPARQL te maken moet meerdere functionaliteiten voorzien worden. Hierboven zijn slechts een (relatief klein) deel van de functionaliteiten beschreven om een beeld te schetsen van de mogelijkheden met SPARQL. Ook de beschreven delen zijn slechts zeer beperkt uitgelegd, om een minimaal beeld te schetsen van de omvang. Het belang van de werking van SPARQL is dat GeoSPARQL een uitbreiding hierop is, dus is er een correcte implementatie nodig van SPARQL alvorens GeoSPARQL geïmplementeerd kan worden. Voor de volledige en uitgebreide uitleg van SPARQL te lezen wordt best doorverwezen naar de officiële documentatie¹.

¹<https://www.w3.org/TR/sparql11-query/>

2.5 Comunica

Comunica is een modulaire SPARQL *query engine* voor het web, gemaakt door het IDLab van de universiteit Gent. Comunica is volledig open source (te vinden op github) en is beschikbaar via de npm *package manager* [20].

2.5.1 Waarom Comunica?

Comunica verschilt van de bestaande *query processors* op verschillende niveau's.

Modulariteit

Dankzij de hoge modulariteit van de Comunica *query engine*, is het mogelijk om uitbreidingen en aanpassingen te doen op de algoritmes en functionaliteiten. Zo kan de gebruiker een op maat gemaakte *engine* maken door de benodigde modules aan elkaar te koppelen aan de hand van een RDF configuratie bestand. Door dit document te publiceren kunnen experimenten ogenblikkelijk gereproduceerd worden door anderen [20].

Heterogene interfaces

Heterogene interfaces binnen Comunica zorgen ervoor dat het mogelijk is om gefedereerd te queryen over heterogene bronnen. Zo wordt het bijvoorbeeld mogelijk om queries over eender welke combinatie van SPARQL *endpoints*, TPF *interfaces*, *datadumps* of andere types *interfaces* te evalueren [20].

Webgebaseerde technologieën

Comunica is geïmplementeerd in JavaScript (of meer specifiek TypeScript) met behulp van web gebaseerde technologieën, specifiek is het geïmplementeerd als een collectie van *Node modules*. Hierbovenop heeft Comunica een *test coverage* van 100% in alle modules. Hierdoor is het mogelijk om Comunica te gebruiken in zowel browsers, de *command line*, het SPARQL protocol als in gelijk welke web- of JavaScript-applicatie [20].

2.5.2 Design patterns

Om het modulaire ontwerp van Comunica mogelijk te maken, zijn er verschillende ontwerppatronen gebruikt. De drie belangrijkste zullen hieronder kort besproken worden.

Publish-subscribe pattern

Het “publish-subscribe” patroon werkt aan de hand van messages tussen de *publishers* en de *subscribers*. Dit patroon doet sterk denken aan “observer”, waarbij alle observerende entiteiten een bericht krijgen wanneer iets veranderd is van het *subject* waar ze naar luisteren. Bij “publish-subscribe” zullen de *publishers* de berichten vrijgeven naar bepaalde categorieën. De *subscribers* kunnen zich dan inschrijven voor deze categorieën, waardoor ze de gepubliceerde berichten kunnen ontvangen zonder kennis te hebben over de *publishers*. Het grote verschil is dan ook onmiddellijk de reden waarom er bij Comunica gekozen is voor “publish-subscribe”. “Publish-subscribe” zorgt voor extra ontkoppeling tussen de verschillende software componenten waarbij er enkel kennis van de categorieën nodig is. In Comunica wordt dit patroon gebruikt om verschillende implementaties toe te staan voor bepaalde taken [20].

Actor model

Het “actor” model is ontwikkeld om zeer parallele systemen te bekomen. Deze systemen bestaan uit verschillende onafhankelijke agents die onderling communiceren aan de hand van *messaging*. Dit is dus gelijkaardig aan het “publish-subscribe” patroon. Hierbij is een *actor* een computationele eenheid die een specifieke taak uitvoert, die reageert op berichten en die berichten kan sturen naar andere *actors*. Het voordeel hiervan is dat *actors* onafhankelijk van elkaar gemaakt kunnen worden om een specifieke taak te voltooien en dat deze asynchroon afgehandeld kan worden. Zo gebruikt Comunica ook het “actor” model om te werken naar de hoge modulariteit. De combinatie met het “publish-subscribe” patroon zorgt ervoor dat elke implementatie van een bepaalde taak hoort bij een aparte *actor* [20].

Mediator pattern

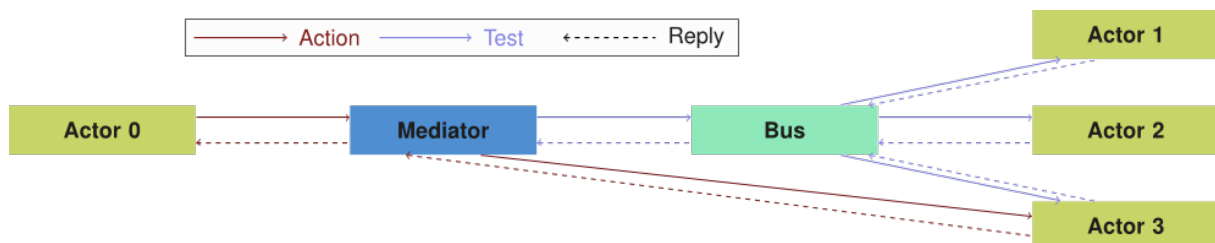
Het “mediator” patroon zorgt voor een verdere reductie van de koppeling tussen software componenten die met elkaar interageren. Dankzij het “mediator” patroon is het ook makkelijk om de interactie tussen deze componenten aan te passen. Dit is mogelijk door het inkapselen van de interactie tussen de software componenten in een *mediator* component. De software componenten zullen nu, in plaats van met elkaar te interageren, communiceren door de *mediator*. Zo

zijn deze componenten autonoom. Verschillende implementaties van deze *mediators* zorgen voor verschillende interactieresultaten. Binnen Comunica wordt dit patroon gebruikt wanneer verschillende *actors* dezelfde taak kunnen oplossen, om te beslissen welke *actor* het meest geschikt is voor een taak. Eventueel kan er zelfs gekozen worden om resultaten van *actors* te combineren tot één oplossing [20].

2.5.3 Architectuur

Het is belangrijk om te weten dat er geen vaste “Comunica engine” bestaat. Comunica is namelijk een *meta engine* die geïntanceerd kan worden in verschillende *engines* gebaseerd op verschillende configuraties. Deze aanpasbaarheid wordt gerealiseerd *at design-time*, gebruikmakend van *dependency injection* [20].

Daarnaast is er ook een enorme flexibiliteit *at run time*. Dankzij de “publish-subscribe”, “actor” en “mediator” patronen kunnen de componenten met elkaar interageren. Dit uit zich in het “Actor-Mediator-Bus” patroon dat gebruikt wordt in Comunica. Dit patroon is te zien in Figuur 2.4. Hierin is te zien hoe een *actor* een actie moet ondernemen. Deze stuurt hij vervolgens door naar de *mediator*. Deze *mediator* zal vervolgens een testactie sturen naar de *bus*. Deze bus zal deze testactie doorsturen naar alle *actors* die geregistreerd staan bij deze *bus*. De *bus* zal dan alle resultaten van deze testactie terugsturen naar de *mediator*, waarop deze zal beslissen welke *actor* het meest geschikt is voor de actie. De uiteindelijk gekozen *actor* zal dan ten slotte de actie uitvoeren, zodat de *mediator* het finale resultaat terug kan sturen naar de *actor* die dit gehele proces heeft gestart [20].



Figuur 2.4: Actor-Mediator-Bus patroon, foto van “Comunica: a Modular SPARQL Query Engine for the Web” [20].

2.5.4 Conclusie

Om tot een besluit te komen over Comunica kunnen er enkele feiten vastgesteld worden. Eerst en vooral kan er veel uitleg gegeven worden, maar is er gepoogd om de werking ervan duidelijk te maken op een korte, doch krachtige manier. Om de volledige en uitgebreide werking en analyse van Comunica te lezen kan best doorverwezen worden naar de officiële paper: “Comunica: a Modular SPARQL Query Engine for the Web”. De belangrijkste punten om te onthouden zijn de vijf doelen die vooropgesteld werden bij het ontwerp van Comunica:

1. **SPARQL query evaluation:** het moet mogelijk zijn om SPARQL queries op een correcte manier te interpreteren en een resultaat weer te geven.
2. **Modulair:** Nieuwe functionaliteiten (of bestaande functionaliteiten aanpassen) zouden slechts een minimale aanpassing aan bestaande code mogen vereisen. Hierbij kan de gebruiker zelf kiezen welke modules hij wil inpluggen voor zijn persoonlijke *engine*.
3. **Heterogene interfaces:** de mogelijkheid om te queryen naar verschillende soorten bronnen (zoals TPF *interfaces*, SPARQL *endpoints* en data dumps in RDF) moet mogelijk zijn.
4. **Federation:** Het moet mogelijk zijn om gefedereerd te queryen. Dit betekent queryen naar verschillende bronnen. In samenhang met de heterogene *interfaces* betekent dit dus queryen naar verschillende bronnen die mogelijks een verschillende soort *interface* hebben.
5. **Web gebaseerd:** Comunica is gemaakt met webtechnologieën zoals Javascript en RDF configuratie bestanden. Hierdoor kan Comunica werken in omgevingen zoals *web browsers*, *local* en zelfs in de *command-line interface*.

2.6 OGC

Voorheen werden bestaande technieken, waarop verder gewerkt wordt, beschreven. Het vervolg van deze masterproef gaat echter over het geospatiale. Dit zijn technieken die gerespecteerd en toegepast moeten worden om zo het uiteindelijke onderzoek te kunnen doen.

OGC staat voor Open Geospatial Consortium. Dit is een wereldwijde *community* die zich inzet voor het verbeteren van de manier hoe omgegaan wordt met geospatiale locatie informatie te verbeteren. Het OGC maakt standaarden om geospatiale informatie beschikbaar te stellen, zodat deze door gebruikers op een optimale en zo uniform mogelijke manier bereikt kan worden [21].

Het OGC voorziet vele standaarden. De standaarden die relevant zijn voor deze masterproef worden hieronder besproken.

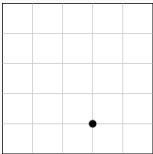
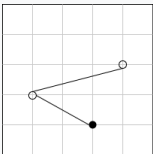
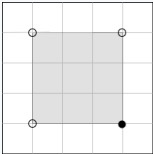
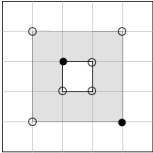
2.6.1 WKT

WKT staat voor *Well-known text*. Dit is een opmaaktaal om de geometrieobjecten op een kaart te representeren. Hierbij worden de coördinaten van een positie gescheiden door een spatie (eerst de x coördinaat, daarna de y coördinaat), opeenvolgende posities binnen één structuur worden gescheiden door een komma [21].

De primitieve geometrieën zijn “*Point*”, “*LineString*” en “*Polygon*”. Een “*Point*” staat voor één exacte locatie op een kaart. Een “*LineString*” gaat dan weer over een lijn, dit zijn dus de verbindingen tussen verschillende punten (bijvoorbeeld van punt 1 naar punt 2 en van punt 2 naar punt 3). Ten slotte is er een “*Polygon*”, wat staat voor een vlak. Hierbij heeft het OGC nog enkele andere voorwaarden opgesteld. Een “*Polygon*” moet topologisch gesloten zijn, wat betekent dat het laatste punt hetzelfde moet zijn als het eerste. Daarbovenop kan een “*Polygon*” bestaan uit een buitenste en binnenste lineaire ring. De buitenste ring slaat op het vlak, terwijl de binnenste ring slaat op een vlak dat uit het grotere vlak gehaald wordt. Hierbij stelt het OGC dat de locaties van de buitenste ring in tegenwijzerszin gegeven moeten zijn, terwijl deze van de binnenste ring in wijzerszin moeten staan [21].

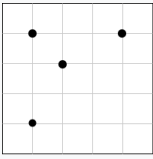
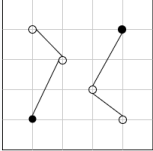
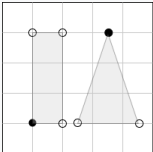
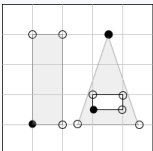
Een verduidelijking van primitieve geometrieën is te zien in Tabel 2.1. Hierin is het eerste punt van een figuur steeds opgevuld voor verduidelijking. Bij een “*Polygon*” met twee ringen wordt ook direct duidelijk waarom de dubbele haken er staan.

Naast de primitieve geometrieën zijn er ook de meerdelige geometrieën. Hierin bestaan “*MultiPoint*”, “*MultiLineString*” en “*MultiPolygon*”. Deze staan respectievelijk voor één of meer van de overeenkomstige primitieve geometrieën. Ter verduidelijking van de meerdelige geometrieën is er ook een klein voorbeeld voorzien in Tabel 2.2. Hierbij gelden uiteraard dezelfde conventies

Type	Example	
Point		POINT (30 10)
LineString		LINESTRING (30 10, 10 20, 40 30)
Polygon	 	POLYGON ((40 10, 40 40, 10 40, 10 10, 40 10)) POLYGON ((40 10, 40 40, 10 40, 10 10, 40 10), (20 30, 30 30, 30 20, 20 20, 20 30))

Tabel 2.1: Primitieve geometrieën.

als bij de primitieve geometrieën.

Type	Example	
MultiPoint		MULTIPOINT ((10 40), (10 10), (20 30), (40 40))
MultiLineString		MULTILINESTRING ((10 10, 20 30, 10 40), (40 40, 30 20, 40 10))
MultiPolygon	 	MULTIPOLYGON (((10 10, 20 10, 20 40, 10 40, 10 10)), ((35 40, 25 10, 45 10, 35 40))) MULTIPOLYGON (((10 10, 20 10, 20 40, 10 40, 10 10)), ((35 40, 25 10, 45 10, 35 40), (30 15, 30 20, 40 20, 40 15, 30 15)))

Tabel 2.2: Meerdelige geometrieën.

2.6.2 GML

GML staat voor *Geography Markup Language*. GML is een XML syntax die gedefinieerd is door het OGC met als doel om geospatiale informatie uit te drukken. GML doet dus hetzelfde als WKT, maar met andere notaties. Het is wel opmerkelijk dat het bij GML enkel mogelijk is om primitieve geometrieën te beschrijven, dus geen meerdelige geometrieën. Aangezien deze standaard minder vaak gebruikt wordt, zal deze ook minder gedetailleerd besproken worden. Zo zijn er drie verschillende manieren om coördinaten weer te geven in GML [21]:

- “<gml:coordinates>”: bij deze *tag* worden de coördinaten gescheiden door een komma. Indien er meerdere locaties na elkaar komen worden deze dan weer gescheiden door een spatie.
- “<gml:pos>”: bij deze *tag* worden de coördinaten gescheiden door een spatie. Hier is het niet mogelijk om meerdere locaties na elkaar te laten komen.
- “<gml:posList>”: deze *tag* is gelijkaardig aan de “pos” *tag*, maar hier is het wel mogelijk om meerdere locaties na elkaar te laten komen. Deze worden dan opnieuw gescheiden door een spatie.

Een kort voorbeeld van de “posList” notatie is te zien in Codefragment 2.8. Hierin wordt het voorbeeld van “*LineString*” uit Tabel 2.1 herschreven naar de GML notatie. Hierbij van het “srsDimension” attribuut op. Dit geeft aan hoeveel dimensies het punt heeft.

```
<gml:LineString gml:id="p21"
  srsName="http://www.opengis.net/def/crs/EPSG/0/4326">
  <gml:posList srsDimension="2">30 10 10 20 40 30</gml:posList>
</gml:LineString >
```

Codefragment 2.8: Voorbeeld GML bij LineString.

2.6.3 GeoSPARQL

Ook GeoSPARQL is een standaard van het OGC. GeoSPARQL is gemaakt voor het representeren en queryen van geospatiale data op het semantische web. Zo definieert GeoSPARQL een vocabulair voor het representeren van geospatiale gegevens in RDF. Verder is het belangrijk te weten dat GeoSPARQL een uitbreiding is op de SPARQL query taal, met als doel het verwerken van geospatiale gegevens. GeoSPARQL is gemaakt om zowel systemen die gebaseerd zijn op kwalitatieve spatiale redenering als systemen die gebaseerd zijn op kwantitatieve spatiale berekeningen te huisvesten. Kortom zal GeoSPARQL nieuwe filterfuncties definiëren voor *Geographic Information System* (GIS) queries (GIS is een informatiesysteem waarmee ruimtelijke gegevens over geografische objecten kunnen opgeslagen, beheerd, bewerkt, geanalyseerd en gepresenteerd worden), gebruikmakend van standaarden die gedefinieerd zijn door het OGC. Dit is echter een zeer korte en beperkte uitleg over GeoSPARQL, de gedetailleerde specificatie worden toegelicht in Sectie 2.7.

2.7 GeoSPARQL

Zoals eerder vermeld is GeoSPARQL één van de vele OGC standaarden. Specifieker is GeoSPARQL uitermate geschikt voor het uitvoeren van GIS queries, maar dit is zeker niet de enige mogelijkheid. Zoals eerder vermeld brengt GeoSPARQL een vocabulair voor de representatie van geospatiale gegevens, gebruik makend van RDF en is GeoSPARQL een uitbreiding op de SPARQL query taal. Het doel van GeoSPARQL is dan ook om geospatiale gegevens te verwerken. Voor het bekomen van een correcte implementatie van GeoSPARQL, zijn er normaliter meerdere voorwaarden waaraan de implementatie moet voldoen. De meeste van deze vereisten maken gebruik van de “geo”, “geof” of “geor” ontologie [21].

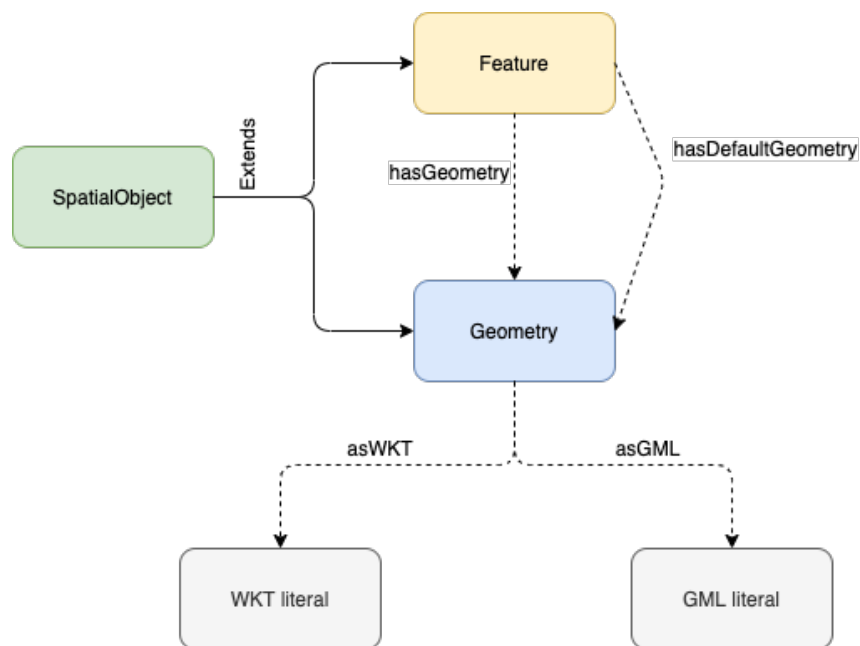
2.7.1 Vereisten

De vereisten zijn opgelijst in de officiële documentatie. Een samenvatting van het geheel is te vinden hieronder [21].

1. **SPARQL:** Deze vereiste is meteen ook de belangrijkste. Deze stelt dat een werkende implementatie van SPARQL aanwezig moet zijn. Dit betekent dat een implementatie GeoSPARQL ook alle mogelijkheden van SPARQL moet voorzien.
2. **Klassen:** een correcte implementatie van GeoSPARQL moet de RDFS klassen “geo:SpatialObject”, “geo:Feature” en “geo:Geometry” toestaan.
3. **Properties:** verder moet een implementatie van GeoSPARQL ook verschillende attributen, zoals “geo:sfContains”, “geo:hasGeometry” en “geo:dimension” naast vele andere, toestaan.
4. **WKT:** RDFS *literals* van het type “geo:wktLiteral” bevatten mogelijk een URI die het coördinaat stelsel van deze coördinaten beschrijft. Indien deze URI niet aanwezig is wordt er gekozen voor “<http://www.opengis.net/def/crs/OGC/1.3/CRS84>”. Daarnaast moeten de coördinaten geïnterpreteerd worden zoals beschreven in het referentiesysteem.
5. **GML:** naast WKT moet GML ook ondersteund worden. De implementatie moet zelf documenteren welke profielen ondersteund worden.
6. **Functies:** Bij de implementatie van GeoSPARQL moeten meerdere functies ondersteund worden. Hierin is onderscheid gemaakt tussen topologische functies (zoals onder andere “geof:sfContains” en “geof:ehContains”) en niet-topologische functies (zoals “geof:distance” en “geof:union”).

7. **Entailment:** GeoSPARQL moet dezelfde semantiek voor *basic graph pattern matching* hanteren als SPARQL. Dit houdt in dat een functie als predicaat moet kunnen omgezet worden naar een query die ook functies berekent om aan een correct resultaat te komen. Hiervoor gebruikt het regels zoals onder andere “geor:sfContains”.

2.7.2 Architectuur



Figuur 2.5: Vereenvoudigd diagram van de GeoSPARQL klassen “Feature” en “Geometry” met sommige properties (figuur gebaseerd op [22]).

Om de architectuur vereenvoudigd weer te geven kan Figuur 2.5 helpen. Hierin is te zien dat er één hoofdklasse is waar de andere van overerven, genaamd “SpatialObject”. Het belangrijke hier is het verschil tussen een “Feature” en een “Geometry”. Een “Geometry” is het effectieve spatiale object, dat weergegeven kan worden als WKT of als GML (zie verder in Subsectie 2.7.3). Eén van de belangrijke (en niet vanzelfsprekende) ontwerpkeuzes is het gebruik van de “Feature” klasse. Deze werkt als een soort wrapper klasse voor “Geometry”. Dit is belangrijk om te weten voor verderop in Subsectie 2.7.6, maar daar zal nogmaals terug verwezen worden naar deze subsectie.

2.7.3 Properties

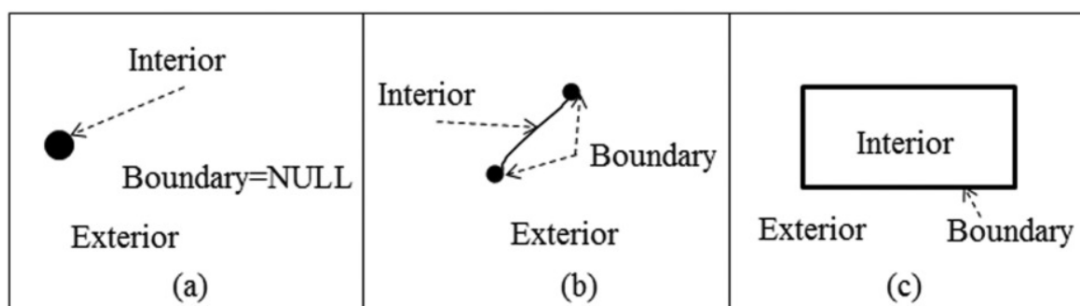
GeoSPARQL voorziet een hele lijst met properties die voorzien moet worden. Vanwege de eerder korte periode om een volledige implementatie van GeoSPARQL te maken, wordt hier minder aandacht aan gegeven. Enkele voorbeelden van deze attributen zijn de volgende [21]:

- **geo:isEmpty**: dit attribuut zal “true” terug geven indien de geometrie geen punten bevat.
- **geo:isSimple**: Dit attribuut zal “true” geven indien de geometrie geen intersecties met zichzelf bevat. Dit kan wel uitgezonderd zijn *boundary* zijn.
- **geo:hasSerialization**: Dit attribuut wordt gebruikt om een geometrie te connecteren met zijn text-gebaseerde serialisatie. Dit kan WKT of GML zijn, maar aangezien deze uitgebreid besproken zijn in Sectie 2.6, wordt dit hier niet herhaald. Het kan wel nogmaals benadrukt worden dat deze *literal* optioneel kan bijhouden welk topologisch referentiesysteem gebruikt wordt. Het is wel belangrijk te weten waarom deze referentiesystemen zo belangrijk zijn. Australië verdriift bijvoorbeeld jaarlijks wat van zijn oorspronkelijke locatie. Na een aantal jaar zou elk huis op de kaart een volledig huis verder liggen, waardoor de informatie nutteloos geworden is. GeoSPARQL lost dit op met het gebruik van de referentiesystemen.

2.7.4 Topologische relaties

Om topologische relaties te kunnen beschrijven, wordt er gebruik gemaakt van drie relatiefamilies. Deze relatiefamilies beschrijven grotendeels gelijkaardige topologische relaties, maar met licht verschillende specificaties. De drie relatiefamilies zijn “Simple Features (sf)”, “Egenhofer (eh)” en “RCC8”. Om de spatiale relaties te beschrijven wordt gebruik gemaakt van een “DE-9IM” patroon. Alvorens de relatiefamilies uitgelegd kunnen worden, is een volledig begrip van de DE-9IM notatie noodzakelijk.

DE-9IM

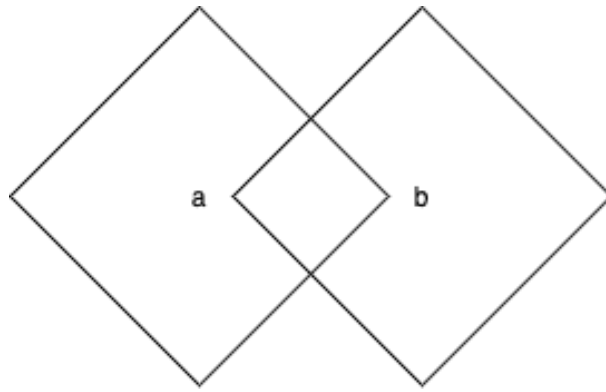


Figuur 2.6: Spatiale objecten met hun *interior*, *boundary* en *exterior*: (a) Een punt; (b) Een lijn; (c) Een vlak. Figuur van [23].

DE-9IM staat voor *Dimensionally Extended Nine-Intersection Model*. Dit model kan afgebeeld worden als een 3x3 matrix, waarbij (in deze volgorde) de *interior*, *boundary* en *exterior* van het

ene spatiale object vergeleken wordt met deze van het andere spatiale object. Deze matrix geeft de dimensies van de intersectie van deze objecten weer. De betekenis van *interior*, *boundary* en *exterior* voor de verschillende soorten spatiale objecten is verduidelijkt in Figuur 2.6. De volledige 3x3 matrix voor de objecten a en b is van de volgende vorm [23]:

$$DE - 9IM(a, b) = \begin{bmatrix} \dim(I(a) \cap I(b)) & \dim(I(a) \cap B(b)) & \dim(I(a) \cap E(b)) \\ \dim(B(a) \cap I(b)) & \dim(B(a) \cap B(b)) & \dim(B(a) \cap E(b)) \\ \dim(E(a) \cap I(b)) & \dim(E(a) \cap B(b)) & \dim(E(a) \cap E(b)) \end{bmatrix}$$



Figuur 2.7: Voorbeeld voor de DE-9IM matrix.

Wanneer dit toegepast wordt op de objecten die te zien zijn in Figuur 2.7, dan wordt de volgende matrix bekomen:

$$DE9IM(a, b) = \begin{bmatrix} 2 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 2 \end{bmatrix}$$

Voor het bepalen van de relaties van de GeoSPARQL families zijn er echter nog iets andere regels opgesteld. Hier heeft het OGC nieuwe betekenissen geïntroduceerd:

1. **Empty:** dit wordt genoteerd als -1.
2. **True:** Dit geldt wanneer de waarden 0, 1, 2 voorkomen. Bij deze waarde kan soms gespecificeerd worden dat het een bepaalde waarde exact moet uitkomen. Het kan ook genoteerd worden als “T” (dit is trouwens meer wel dan niet het geval).
3. **False:** dit geldt wanneer de waarde -1 wordt uitgekomen, maar wordt genoteerd als “F”.
4. **Don’t care:** Dit betekent dat eender welke waarde mag voorkomen en dat dit hier niet naar gekeken moet worden. Dit wordt genoteerd als “*”.

Hierbij wordt het patroon geschreven als een sequentie van negen karakters. Als deze van links naar rechts gelezen wordt, dan stelt dit de DE-9IM matrix voor van linksboven beginnend, rij per rij opvullend. Het volgende patroon staat dus voor de overeenkomstige matrix:

$$T ** T ** T ** = \begin{bmatrix} T & * & * \\ T & * & * \\ T & * & * \end{bmatrix}$$

Indien er meer dan één lijn staat, wil dit zeggen dat één van de mogelijke matrices voldoende is om te besluiten dat de relatie klopt.

Ten slotte kunnen ook niet alle relaties voorkomen bij elke object-combinatie (bijvoorbeeld twee punten kunnen elkaar niet kruisen). Daarom wordt er ook een nieuwe notatie toegevoegd in de GeoSPARQL-documentatie, om te vermelden op welke spatiale objecten het DE-9IM intersectiepatroon van toepassing is. Hierbij staat symbool “P” voor 0-dimensionale geometrieën (zoals punten). Het symbool “L” staat dan weer voor 1-dimensionale geometrieën (zoals lijnen). Het laatste symbool is “A”, wat dan weer staat voor 2-dimensionale geometrieën (zoals vlakken) [21].

Simple Features

De eerste relatiefamilie is de “Simple Feature” family. De regels van deze familie worden weergegeven in Tabel 2.3. Hierin is exact te zien hoe de implementaties verwacht worden te werken. Om de grootte van deze masterproef in te perken is enkel deze relatiefamilie uitgewerkt en wordt dus ook enkel deze relatiefamilie besproken. Indien het voorbeeld van de “contains” functie besproken wordt, moet dit als volgt geïnterpreteerd worden: “a contains b is waar, indien de intersectie van hun *interiors* bestaat en hierbovenop de intersectie van de *exterior* van a met zowel de *interior* als de *boundary* van b niet bestaat”. Dit betekent dus dat b in a ligt en dat geen enkel deel van b buiten a ligt [21].

Relation Name	Relation URI	Domain/Range	Applies To Geometry Types	DE-9IM Intersection pattern
equals	geo:sfEquals	geo:SpatialObject	All	(TFFFTFFFT)
disjoint	geo:sfDisjoint	geo:SpatialObject	All	(FF*FF****)
intersects	geo:sfIntersects	geo:SpatialObject	All	(T***** *T***** ***T***** ****T****)
touches	geo:sfTouches	geo:SpatialObject	All except P/P	(FT***** F**T***** F***T****)
within	geo:sfWithin	geo:SpatialObject	All	(T*F**F***)
contains	geo:sfContains	geo:SpatialObject	All	(T*****FF*)
overlaps	geo:sfOverlaps	geo:SpatialObject	A/A, P/P, L/L	((T*T***T**) for A/A, P/P; (1*T***T**) for L/L)
crosses	geo:sfCrosses	geo:SpatialObject	P/L, P/A, L/A, L/L	((T*T***T**) for P/L, P/A, L/A; (0*****)) for L/L)

Tabel 2.3: Simple Features topologische relaties (tabel van [21]).

2.7.5 Niet-topologische relaties

Verder zijn er ook de niet-topologische relaties. Verschillende van deze functies gebruiken een meeteenheid-URI. Daarom heeft het OGC enkele standaard meeteenheden gedefinieerd, te vinden onder de *namespace* “<http://www.opengis.net/def/uom/OGC/>”. Een voorbeeld van een dergelijke meeteenheid is “[<http://www.opengis.net/def/uom/OGC/metre>](http://www.opengis.net/def/uom/OGC/metre)”. Enkele van de niet-topologische functies zijn de volgende: [21].

- **geof:distance**: deze functie moet de kortst mogelijke afstand geven tussen twee objecten.
- **geof:buffer**: deze functie geeft een geometrie terug die alle punten bevat die binnen een bepaalde radius van een meegegeven geometrie liggen.
- **geof:convexHull**: deze functie geeft een geometrie terug die het kleinste convexe omhulsel

is dat alle punten omvat van een meegegeven geometrie.

- **geof:intersection:** deze functie geeft een geometrie terug die staat voor alle punten van de intersectie tussen twee geometrieën.
- **geof:union:** deze functie geeft een geometrie terug die staat voor alle punten in de unie van twee geometrieën.
- **geof:envelope:** Deze functie geeft de minimaal omvattende (*bounding*) *box* weer voor een geometrische figuur. Dit betekent dus de kleinst mogelijke rechthoek waar elk punt van de geometrie in zit.

Ten slotte is het belangrijk te vermelden dat alle berekeningen horen te gebeuren in het referentiesysteem van de eerste geometrie die meegegeven is aan een functie, zowel bij topologische als niet-topologische functies [21].

2.7.6 Query Rewrite Extension

De laatste uitdaging is het probleem waar de topologische functies als predicaat staan. Hierbij zou verwacht worden dat deze op voorhand berekend zijn en zo opgeslagen zijn in de RDF dataset. Hierbij is echter het probleem dat wanneer dit niet op voorhand berekend is, men nog steeds de correcte oplossing wil. Wanneer een deel van de gegevens uit de ene dataset komt en het andere deel uit een andere bron, zelfs dan wordt een correct antwoord verwacht, hoewel hier geen optie is tot het op voorhand uitrekenen [21].

Hiervoor wordt gebruik gemaakt van de *query rewrite* uitbreiding. Deze techniek zal de query uitbreiden, door de unie (de SPARQL *union*, niet de niet-topologische functie van GeoSPARQL!) te nemen van de oorspronkelijke stelling met enerzijds de relatie als predicaat en anderzijds het nieuwe uitgebreide deel waarbij dezelfde relatie uitgerekend wordt als functie. Hierbij is het ook belangrijk rekening te houden met het verschil tussen een “Feature” en een “Geometry” (zoals uitgelegd in Subsectie 2.7.2), waardoor er vier extra delen nodig zijn voor de mogelijke combinaties tussen “Feature” en “Geometry” [21].

Zo heeft het OGC een template gemaakt om dit te herschrijven. Bij deze template zijn er een aantal placeholders gebruikt [21]:

- **ogc:relation:** deze placeholder staat voor de relatie die gebruikt wordt (dit zou bijvoorbeeld de “geo:sfContains” kunnen zijn).
- **ogc:function:** deze placeholder staat voor de overeenkomstige functie bij de relatie die gebruikt wordt (in het voorbeeld van “geo:sfContains” zou de functie “geof:sfContains” zijn).

- **ogc:asGeomLiteral**: deze placeholder staat voor één van de serialisatie technieken om het object te bekomen.

Een voorbeeld voor het herschrijven van een query is te vinden in Codefragment 2.9, maar het uiteindelijke template voor het herschrijven van de query is te vinden in Codefragment 2.10.

```
select *
where {
  { ?f1 ogc:relation ?f2 . }
}
```

Codefragment 2.9: Voorbeeldquery die herschreven zal worden.

```
select *
where {
  { ?f1 ogc:relation ?f2 . }
  UNION
  # feature - feature rule
  {
    ?f1 geo:hasDefaultGeometry ?g1 .
    ?f2 geo:hasDefaultGeometry ?g2 .
    ?g1 ogc:asGeomLiteral ?g1Serial .
    ?g2 ogc:asGeomLiteral ?g2Serial .
    filter(ogc:function(?g1Serial, ?g2Serial)) }
  UNION
  # feature - geometry rule
  {
    ?f1 geo:hasDefaultGeometry ?g1 .
    ?g1 ogc:asGeomLiteral ?g1Serial .
    ?f2 ogc:asGeomLiteral ?g2Serial .
    filter(ogc:function(?g1Serial, ?g2Serial)) }
  UNION
  # geometry - feature rule
  {
    ?f2 geo:hasDefaultGeometry ?g2 .
    ?f1 ogc:asGeomLiteral ?g1Serial .
    ?g2 ogc:asGeomLiteral ?g2Serial .
    filter(ogc:function(?g1Serial, ?g2Serial)) }
  UNION
  # geometry - geometry rule
  {
    ?f1 ogc:asGeomLiteral ?g1Serial .
    ?f2 ogc:asGeomLiteral ?g2Serial .
    filter(ogc:function(?g1Serial, ?g2Serial)) }
}
```

Codefragment 2.10: Template om queries te herschrijven (codefragment van [21]).

“Programming isn’t about what you know; it’s about what you can figure out.”

~Chris Pine

3

Implementatie

Om de hypothesen uit Sectie 1.3 af te toetsen, moet er een werkende implementatie zijn van GeoSPARQL. De structuur van dit hoofdstuk zal dezelfde volgorde aannemen als hoe de implementatie is verlopen. Dit vertoont een structuur die toelaat dat elk deel steeds onmiddellijk getest kan worden.

3.1 Comunica

Zoals eerder aangehaald speelt Comunica een zeer belangrijke rol bij deze implementatie van GeoSPARQL (zie Sectie 2.5). Hierbij (dankzij de modulariteit) is het mogelijk om een actor aan te maken voor het afhandelen van GeoSPARQL-functionaliteiten. Deze actor maakt gebruik van “sparqlalgebrajs” om de SPARQL query om te vormen naar SPARQL algebra en van “sparqlee” om deze SPARQL algebra correct uit te werken. De gemaakte implementatie in Comunica is terug te vinden op GitHub¹.

3.1.1 Sparqlalgebrajs

¹<https://github.com/dreeki/comunica>

```

SELECT ?f
WHERE {
  my:A my:hasExactGeometry ?aGeom .
  ?aGeom geo:asWKT ?aWKT .
  ?f my:hasExactGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .
  FILTER (geof:sfContains(?aWKT, ?fWKT) && !sameTerm(?aWKT, ?fWKT))
}

```

Codefragment 3.1: Voorbeeld van GeoSPARQL query.

Sparqlalgebra's zorgt er onder andere voor dat de filter omgezet wordt naar een boomstructuur die recursief kan worden doorlopen. Deze omvorming is te zien in Codefragment 3.1 en Codefragment 3.2. In Codefragment 3.1 is een voorbeeld van een correcte GeoSPARQL query te zien (ter informatie: deze query haalt alle vormen op, die geografisch in “my:A” liggen, maar niet “my:A” zelf zijn), maar voor het uitvoeren van deze query moet deze eerst omgevormd worden naar SPARQL algebra. Vanwege de grootte van het resultaat is slechts de essentie hiervan terug te vinden in Codefragment 3.2. Aangezien het gaat over de filterfunctie, is enkel dat deel terug te vinden. Hierbij is zeer duidelijk de recursieve boomstructuur terug te vinden, waarbij in de wortel van de boom de operator “&&” te zien is. Dit blad in de boom zal vervolgens een linker- en rechterkind hebben. Deze zijn te vinden in het veld “args”. Elk blad zal zo recursief een waarde krijgen die berekend wordt uit de bladeren die zich lager in deze boom bevinden. In dit voorbeeld zijn dit de “sfContains” functie aan de linkerhelft en de “!” operator aan de rechterhelft. Dit wordt op deze manier recursief uitgevoerd.

```

{
  ...,
  expression: {
    type: 'expression',
    expressionType: 'operator',
    operator: '&&',
    args: [{
      type: 'expression',
      expressionType: 'named',
      name: NamedNode {
        id: 'http://www.opengis.net/def/function/geosparql/sfContains'
      },
      args: [{
        type: 'expression',
        expressionType: 'term',
        term: Variable { id: '?aWKT' }
      },
      {
        type: 'expression',
        expressionType: 'term',
        term: Variable { id: '?fWKT' }
      }
    ]
  },
  {
    type: 'expression',
    expressionType: 'operator',
    operator: '!',
    args: [{
      type: 'expression',
      expressionType: 'operator',
      operator: 'sameterm',
      args: [{
        type: 'expression',
        expressionType: 'term',
        term: Variable { id: '?aWKT' }
      },
      {
        type: 'expression',
        expressionType: 'term',
        term: Variable { id: '?fWKT' }
      }
    ]
  }
]}
}

```

Codefragment 3.2: Voorbeeld van SPARQL algebra.

3.1.2 Sparqlee

Sparqlee is een SPARQL *expression evaluator*. Dit betekent dat sparqlee een SPARQL algebra expressie zal evalueren. Sparqlee is bij deze implementatie gebruikt voor het maken van de GeoSPARQL functionaliteiten, omdat sparqlee deze recursieve boomstructuur van sparqlalgebra's, volledig afhandelt voor SPARQL queries. Dit is belangrijk voor de belangrijkste vereiste van GeoSPARQL, namelijk het hebben van een werkende SPARQL implementatie. Sparqlee controleert eerst de volledige expressie om te zien of het geheel verwerkt kan worden. Dit is enkel het geval als sparqlee de verschillende functies en operators correct kan afhandelen. In dit specifieke geval betekent het dus dat sparqlee de GeoSPARQL functies moet kunnen uitvoeren. Hiervoor is dezelfde programmeerstijl gehanteerd als deze die al aanwezig is in sparqlee zelf. De gemaakte implementatie in sparqlee is terug te vinden op GitHub².

3.1.3 Verbeteringen

Een eerste verbetering aan deze eerste keuze is meteen dat deze GeoSPARQL functies niet in sparqlee gemaakt zouden mogen worden. Sparqlee is louter een SPARQL *expression evaluator*, wat betekent dat deze niet meer dan enkel SPARQL hoort te ondersteunen. De oplossing hiervoor is het ondersteunen van *custom functions* binnen sparqlee (dit staat bovendien binnen de specificaties van SPARQL). Indien deze *custom functions* ondersteund zouden worden binnen sparqlee, dan zou het mogelijk zijn om de functies te implementeren binnen de hiervoor gemaakte actor in Comunica. Op deze manier kan deze dan geïnjecteerd worden in sparqlee. Het implementeren van deze *custom functions* functionaliteit binnen sparqlee is echter te complex en tijdrovend voor de eerder kleine impact op deze masterproef. Dit blijft echter wel een vereiste voor de modulariteit, maar dit wordt gezien als *future work*.

²<https://github.com/dreeki/sparqlee>

3.2 Datastructuur

Voordat begonnen kan worden aan de effectieve oplossing van GeoSPARQL functies, moet de gepaste datastructuur gekozen worden om dit te kunnen realiseren. Zoals beschreven in Sectie 2.7.2, moet er ondersteuning zijn voor zowel “Geometry” als “Feature” objecten.

3.2.1 GeoJSON

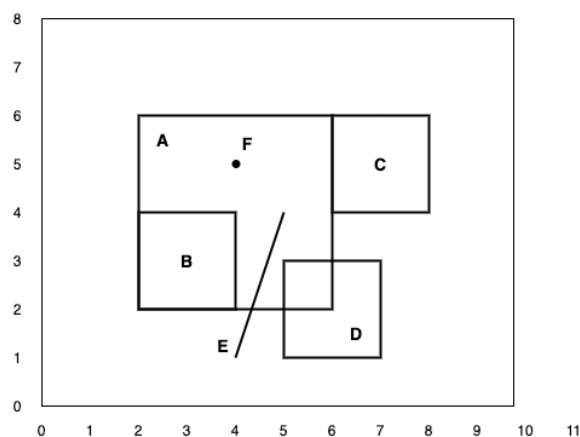
Om dit te kunnen ondersteunen is er gekozen voor een vaker gebruikt formaat, genaamd GeoJSON. GeoJSON biedt ondersteuning voor zowel “Geometry” als “Feature” objecten, waarbij “Feature” objecten een “Geometry” object bevatten, naast andere attributen. Bovendien ondersteunt GeoJSON de volgende vormen: “Point”, “LineString”, “Polygon”, “MultiPoint”, “MultiLineString” en “MultiPolygon”. Op deze manier zijn alle voorwaarden van de architectuur voor GeoSPARQL voldaan. Het volgende probleem is dan: hoe kan een WKT string omgevormd worden naar GeoSPARQL?

3.2.2 Terraformer

Terraformer is een geografische toolkit voor het werken met onder andere geometrieën, geografie en formaten. Verder is Terraformer opgesplitst in enkele modules. Hier wordt verder (zie Sectie 3.3) op terug gekomen. Voorlopig wordt enkel verder ingegaan op de Terraformer WKT parser. Deze module is gemaakt om te voorzien in een eenvoudige omzetting van WKT string naar GeoJSON en indien gewenst ook in de omgekeerde richting. Op deze manier kan een geserialiseerde vorm (namelijk WKT string) omgezet worden naar GeoJSON, zodat nu verder gegaan kan worden met de implementatie.

3.3 Topologische functies

Een van de belangrijkste functionaliteiten van GeoSPARQL is het topologisch van elkaar kunnen onderscheiden van vormen. Hiervoor zijn er drie verschillende relatiefamilies (zoals eerder aangehaald, zie Subsectie 2.7.4). Vanwege de eerder korte periode voor deze masterproef is ervoor gekozen om de focus te leggen op slechts één van deze families, namelijk de “Simple Features” familie. Om niet te vaak in herhaling te vallen kan best terugverwezen worden naar Subsubsectie 2.7.4 om te kijken naar de specificaties van deze familie. Om de eerste functie te maken is gekozen voor de “sfContains” functie. De *contains*-functie is een veel gebruikte en voor de hand liggende functie, die gaat controleren of een vorm binnen een andere vorm ligt. Om de verschillende functies te testen werd een eenvoudige testset gecreëerd. Deze testset is te zien in Figuur 3.1. Om tot een eerste uitkomst te komen, was er al een zeer goede oplossing. Dit bleek Terraformer te zijn.



Figuur 3.1: Illustratie geospatiale data van [21]

3.3.1 Terraformer

Terraformer was een eerste en zeer voor de hand liggende keuze, omdat deze reeds gebruikt werd voor de omzetting van WKT naar GeoJSON. Zo heeft Terraformer een module voor het werken met GeoJSON. De module heet Terraformer Core. Deze module voorziet functies voor onder andere het controleren of een vorm binnen een andere vorm ligt en om te controleren of een vorm een andere vorm snijdt. Dit is welliswaar te beperkt om de volledige “Simple Features” familie te implementeren, maar voor de “sfContains” en nadien voor zowel de “sfIntersects” en de “sfDisjoint” functies is dit zeker een goed begin.

Na het maken van de implementatie, bleken er toch enkele problemen te zijn met de “contains” functie van Terraformer. Zo valt onmiddellijk op dat er vele randgevallen zijn die niet correct

werken. Enkele voorbeelden daarvan zijn de volgende:

- Polygoon “contains” lijn: wanneer de lijn van een hoekpunt naar een ander hoekpunt gaat (dus wanneer het een zijde of een diagonaal is), zou dit “true” moeten geven, maar dit geeft “false”.
- Polygoon “contains” polygoon: wanneer de polygonen een hoekpunt delen (dus deze ligt erin, maar ligt helemaal in de hoek), zou dit “true” moeten geven, maar dit geeft “false”.

Om, gebruik makend van Terraformer, deze problemen op te lossen zou teveel code aangepast moeten worden. Aangezien deze randgevallen zeer vaak voorkomen (en aangezien de kans reëel is dat er nog meerdere andere fouten zijn), werd besloten om een andere oplossing te zoeken die meer bruikbaar is voor het geheel, zoals de andere topologische functies.

3.3.2 Manueel

Het eerstvolgende idee was om dit manueel op te lossen. Dit leek een logische oplossing, omdat zo alle functies met zekerheid gemaakt kunnen worden en dit met een implementatie die conform is met de OGC-standaarden. Hier werd echter vrij snel van afgeweken vanwege de complexiteit. Het uitgevoerde onderzoek wordt hieronder beschreven, opnieuw voor het voorbeeld van de “contains” functie. Hierbij werd ook enkel aandacht gegeven aan de eenvoudige vormen (dus “Point”, “LineString” en “Polygon”). De gedachtengang hierbij was dat de meervoudige vorm identiek was, maar zich meerdere keren herhaalt.

Point

Bij het beginnen van een eigen implementatie is het eenvoudigste om te beginnen bij een punt. Hierbij is het namelijk zo dat een punt enkel in een punt ligt wanneer dit punt exact hetzelfde is. Het controleren of een lijn of een polygoon in een punt ligt is nog eenvoudiger, dit is namelijk niet mogelijk (en bijgevolg dus altijd “false”).

LineString

Het volgende deel is logischerwijs controleren wat binnen een lijn kan liggen. Voor een punt is dit wiskundig eenvoudig op te lossen, door voor elk deel van de lijn de richtingscoëfficiënt uit te rekenen en vervolgens te kijken of het punt op de lijn ligt. Hierbij moet wel opgelet worden, aangezien het deel van de lijn niet oneindig doorloopt. Dit is evenwel eenvoudig op te lossen

door te controleren dat de x-coördinaat van het punt tussen de x-coördinaten van de uiteinden van de lijn ligt.

Bij het controleren of een lijn in een andere lijn ligt, kan dezelfde techniek als bij een punt gebruikt worden. Hierbij moet voor elk deel van de mogelijks binnen liggende lijn gecontroleerd worden of de uiteinden op de eerste lijn liggen. Ook moet er gecontroleerd worden of de richtingscoëfficiënt van beide lijnen in dit deel (het volledige deel) identiek is. Indien deze voorwaarden voldaan zijn, dan ligt de lijn binnen de andere lijn.

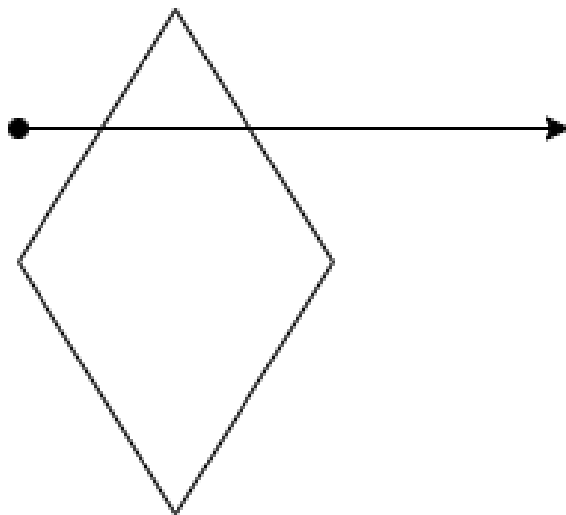
De controle of een polygoon binnen een lijn ligt is overbodig. Dit is namelijk onmogelijk, bijgevolg kan deze functie rechtstreeks “false” als uitkomst geven.

Polygon

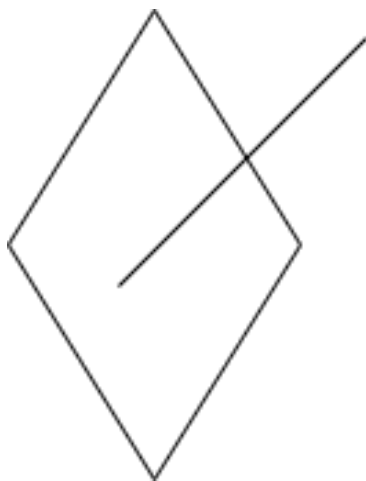
Het laatste en moeilijkste deel is controleren of een vorm binnen een polygoon ligt. Het eerste deel is het controleren of een punt binnen een polygoon ligt. Intuïtief wordt direct gedacht aan het wiskundig beschrijven van een vlak, maar dit is echter niet bruikbaar. Vervolgens kan gecontroleerd worden of het punt onder of boven een lijn ligt, om zo te proberen achterhalen of dit betekent dat het punt binnen de polygoon ligt. Het probleem heeft echter een eenvoudigere oplossing dan dit. Wanneer het punt op een lijn van de polygoon ligt, is meteen geweten dat de “contains” functie “true” moet weergeven. Wanneer dit niet het geval is, kan een andere techniek gebruikt worden. Zo kan een horizontale lijn getrokken worden, vertrekkend vanuit het punt en helemaal naar rechts (met oneindige lengte). Wanneer nu geteld wordt met hoeveel lijnen van de polygoon deze lijn kruist, is de uitkomst gekend. Het punt ligt namelijk binnen de polygoon wanneer het aantal kruissende lijnen een oneven aantal is. Deze techniek is geïllustreerd in Figuur 3.2.

Het volgende deel is het controleren of een lijn binnen een polygoon ligt. Hiervoor is er opnieuw een techniek. Voor de lijn moet gecontroleerd worden of een punt binnen de polygoon ligt. Indien dit het geval is, moet ook nog eens gecontroleerd worden of de lijn snijdt met een rand van de polygoon. Dit betekent dat elke lijn van de rand van de polygoon gecontroleerd moet worden. Het controleren of lijnen snijden is opnieuw wiskundig aan te pakken, maar hier wordt niet verder op ingegaan. Deze techniek is geïllustreerd in Figuur 3.3.

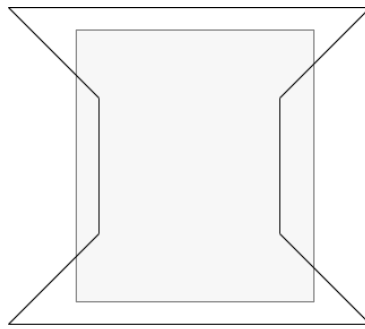
Het laatste deel hierbij is het controleren of een polygoon binnen een polygoon ligt. Hiervoor zou intuïtief gedacht kunnen worden aan het controleren of alle punten van de (vermoedelijk) binnenste polygoon binnen de andere polygoon liggen. Zoals geïllustreerd in Figuur 3.4 is dit niet altijd het geval. Dit is enkel mogelijk indien de buitenste vorm convex (= bolvormig) is. Aangezien dit in vele gevallen niet zo is, zou het verlies van performantie zijn om hierop te controleren. Een correcte techniek is echter controleren of één enkel punt van de (vermoedelijk)



Figuur 3.2: Voorbeeld polygon contains point.



Figuur 3.3: Voorbeeld van polygon contains line.



Figuur 3.4: Voorbeeld van polygon contains polygon.

binnenste polygoon binnen de andere polygoon ligt. Vervolgens volstaat het om te controleren of er een lijn van de rand van de ene polygoon snijdt met een lijn van de rand van de andere polygoon. Indien er snijdende lijnen zijn, zal de “contains” functie “false” teruggeven. Hierbij wordt ook duidelijk dat deze berekeningen computationeel intensief worden, zeker in het geval van queryen. Dit kan nog verbeterd worden door de lijnintersectietests te versnellen met het “*sweep line*” algoritme, maar hier wordt niet verder op ingegaan. In het voorbeeld in Figuur 3.4 is de gekleurde polygoon diegene die getest wordt om binnen de andere te liggen.

Extra moeilijkheden

De eerder vernoemde technieken zijn al vrij ingewikkeld en dit is slechts voor de “contains” functie. Hier zouden nog meerdere andere technieken gebruikt moeten worden om de andere functies te implementeren. Dit wordt nog ingewikkelder wanneer ook rekening gehouden wordt met de veelvoudige vormen zoals “MultiPoint”, “MultiLineString” en “MultiPolygon”. Hierbovenop is er ook nog geen rekening gehouden met de binnenste ring van de polygonen, die een deel van het vlak excluseren. Dit is nog een extra moeilijkheidsfactor waar rekening mee moet worden gehouden.

Zo zijn er vele nadelen aan het zelf implementeren hiervan. Een eigen implementatie bevat snel fouten bij de vereiste functionaliteiten zoals ze hierboven beschreven werden. Ook moet dit uitvoerig getest worden, wat best door een maximaal aantal gebruikers gebeurt. Bovendien moeten de meest performante algoritmes gebruikt worden, zodat dit bruikbaar is voor queries. Al deze redenen verwerpen het zelf implementeren. Daarom werd ervoor gekozen om overgegaan “Turf.js” te gebruiken.

3.3.3 Turf.js

“Turf.js” is een modulaire geospatiale engine die gemaakt is in JavaScript. Zo is Turf een JavaScript *library* die werkt aan de hand van GeoJSON. Het is een verzameling van kleine modules, zodat gebruik kan gemaakt worden van exact wat nodig is voor de *use case*. Turf gebruikt naar eigen zeggen de nieuwste algoritmes, wat een pluspunt is voor de performantie. Bovendien heeft Turf een uitgebreide *community*, wat ervoor zorgt dat mogelijke fouten in de implementatie gevonden en opgelost worden. Zo kan een bug opgelost worden door de nieuwste versie binnen te halen.

Turf voorziet vele methoden om te gebruiken in eigen berekeningen. Daarnaast heeft Turf al enkele *build in* functies, zoals “booleanContains”, “booleanDisjoint” en “booleanOverlap”. Deze functies zijn exact wat nodig is. Een overzicht van de functies die gebruikt zijn voor de implementatie van de “Simple Features” familie is te zien in Tabel 3.1.

GeoSPARQL functie	Turf.js functie	Opmerkingen
sfEquals	booleanEqual	De functie van Turf is volledig conform met de documentatie.
sfDisjoint	booleanDisjoint	De functie van Turf blijkt volledig conform te zijn met de documentatie van GeoSPARQL. Deze bevat echter een bug waarbij twee evenwijdige en overlappende lijnen toch “disjoint” zouden zijn.
sfIntersects	!booleanDisjoint	Deze functie heeft hetzelfde probleem als sfDisjoint, aangezien deze het inverse is.
sfTouches	/	Turf heeft nog geen functie die gebruikt kan worden voor deze functionaliteit.
sfWithin	booleanWithin	De functie van Turf blijkt volledig conform te zijn met de documentatie van GeoSPARQL. Deze bevat echter een bug bij het gebruik van de binnenste ring.
sfContains	booleanContains	De functie van Turf blijkt volledig conform te zijn met de documentatie van GeoSPARQL. Deze bevat echter een bug bij het gebruik van de binnenste ring.
sfOverlaps	booleanOverlap	De functie van Turf is niet volledig conform met de documentatie van GeoSPARQL. Het verschil hierbij is dat het voor Turf voldoende is om enkel de borders te laten overlappen, terwijl GeoSPARQL benadrukt dat ook de interiors moeten overlappen.
sfCrosses	booleanCrosses	De functie “booleanCrosses” van Turf zou overeen moeten komen, maar deze heeft andere specificaties waardoor deze niet bruikbaar is.

Tabel 3.1: Implementatie GeoSPARQL functies (Simple Features familie) met “Turf.js”.

3.4 Niet-topologische functies

De niet-topologische functies binnen GeoSPARQL zijn beperkter in hoeveelheid, ten opzichte van de topologische functies. In plaats van het onderscheiden van vormen zorgen deze functies voor het uitvoeren van berekeningen. Dit kan bijvoorbeeld het opzoeken van de kortste afstand tussen twee vormen zijn of het samenvoegen van twee vormen om vervolgens een topologische functie erop los te laten. Aan deze functies wordt iets minder aandacht besteed dan aan de topologische functies, omdat deze duidelijk minder gebruikt zullen worden. Toch is er een implementatie gemaakt voor de “union” en “intersection” functies. De “union” functie zal twee vormen samenvoegen tot één nieuwe vorm. De “intersection” functie zal dan weer kijken naar het deel dat twee vormen gemeenschappelijk hebben en dit teruggeven.

Voor het maken van deze implementaties is opnieuw gebruik gemaakt van Turf en dankzij de implementatie van sparqlee, die excellent gebruik maakt van het *divide-and-conquer* (= verdeel en heers) programmeer principe, is deze implementatie op bijna identiek dezelfde manier als de topologische functies mogelijk.

3.4.1 Beperkingen

Aangezien gebruik gemaakt wordt van Turf, moet de manier van Turf gevolgd worden. Hierbij is het enkel mogelijk om de unie of intersectie te berekenen van polygonen. GeoSPARQL beschrijft echter dat deze functies beiden een geometrisch object terug moeten geven die alle punten representeert van het resultaat van deze functies. Dit impliceert dat het ook mogelijk moet zijn om deze functies toe te passen op punten en lijnen. Dit zou zelfs mogelijk moeten zijn voor de combinatie van punten, lijnen en polygonen.

Dit laatste is echter niet mogelijk met de huidige implementatie waar gebruik gemaakt wordt van “Point”, “MultiPoint”, “LineString”, “MultiLineString”, “Polygon” en “MultiPolygon”. In Figuur 3.5a is een voorbeeld te zien van twee vormen waarvan de unie (= de vorm die alle



Figuur 3.5: Problematiek union.

punten van beide figuren samen bevat) genomen wordt. In Figuur 3.5b wordt aangetoond dat het resultaat van de unie van een “LineString” en een “Polygon” niet te representeren valt in één van de zes eerder genoemde vormen. Hiervoor zou gewerkt moeten worden met een soort “GeometryCollection”.

3.5 Referentiesysteem

Om een werkende implementatie te bekomen, is er nog een functionaliteit die ondersteund moet worden. Zo moet het mogelijk zijn om te werken met verschillende referentiesystemen. Om nog eens te benadrukken waarom dit belangrijk is, wordt dit geïllustreerd met een voorbeeld. Verschillende landen of zelfs delen van landen liggen op andere aardplaten die los van elkaar bewegen of zelfs tegen elkaar botsen. Hierdoor kunnen landen ten opzichte van elkaar bewegen. Het best voorbeeld hiervan is Australië dat jaarlijks wat kan verplaatsen. Op deze manier zou na enkele jaren elke geografische entiteit opnieuw vastgelegd moeten worden. Hiervoor wordt gebruik gemaakt van aparte referentiesystemen. Dit is een manier om een punt te beschrijven in coördinaten ten opzichte van een relatief punt. Om echter coördinaten in verschillende referentiesystemen te kunnen vergelijken met elkaar, moeten deze eerst omgerekend worden naar hetzelfde referentiesysteem.

Voor het oplossen van dit probleem bestaan twee mogelijkheden. Ofwel worden beide referentiesystemen omgerekend naar één op voorhand gedefinieerd referentiesysteem, ofwel wordt één van beide referentiesystemen naar de andere omgerekend. Er is gekozen voor de tweede optie om twee redenen. In dit geval moet voor elk koppel vormen, dat vergeleken wordt, slechts één vorm herrekend worden, wat zorgt voor een betere performantie. De tweede reden is dat het OGC zelf voorschrijft dat gewerkt moet worden in het referentiesysteem van de eerste vorm.

3.5.1 Proj4js

“Proj4js” is een *library* die zorgt voor het transformeren van coördinaten van het ene referentiesysteem naar coördinaten van het andere referentiesysteem. Hierbij zijn al enkele projecties op voorhand gedefinieerd binnen Proj4, maar het is ook mogelijk om zelf nieuwe projecties toe te voegen. Proj4 is dan ook de gebruikte *library* voor het uitvoeren van deze berekening bij de gemaakte implementatie.

3.5.2 Beperkingen

Bij deze functionaliteit zijn er nog wel enkele beperkingen. Er zijn nog niet veel referentiesystemen beschreven binnen GeoSPARQL, waardoor het slechts in beperkte mate mogelijk is om van deze functionaliteit gebruik te maken. De architectuur van hoe hiermee gewerkt wordt, is echter wel in orde. Kortom betekent dit dat het nog niet zeer handig is om te gebruiken, maar dat het wel klaar is om in de toekomst toegepast te worden.

3.6 Testomgeving

In de voorgaande secties werd beschreven hoe de implementatie van GeoSPARQL gemaakt is. Nu er een werkende implementatie is, is het mogelijk om over te gaan naar de volgende stap. Hier zijn drie doelen, namelijk:


1. Visualiseren van het geheel, met makkelijk aanpasbare queries.
2. Omgeving voor het geven van demonstraties van de implementatie.
3. Omgeving voor het uitvoeren van tests, voor het aftoetsen van de hypothesen.

Om dit te bereiken is gebruik gemaakt van de “jQuery Widget” van Comunica. Deze geeft een grafische *user interface*, die toestaat om te queryen over één of meerdere bronnen. Hierbij is het mogelijk om de bronnen zelf in te geven. Daarnaast is het ook mogelijk om de query zelf in te geven. Deze geeft een overzicht van het resultaat en een “execution log” terug, zodat gecontroleerd kan worden welke stappen doorlopen werden voor het bekomen van het resultaat. Het geheel hiervan is zichtbaar in Figuur 3.6.

Hiermee is het eerste doel onmiddellijk bereikt. Dankzij de visualisatie die zeer gelijkaardig is aan andere gekende *query engines*, is dit zeer geschikt voor het geven van demonstraties. Ten slotte, zoals eerder vermeld, is het mogelijk om zowel de bronnen in te geven als de volgorde van uitvoering te bekijken. Hierdoor is het uitermate geschikt voor het uitvoeren van tests en voor het aftoetsen van de hypothesen.

Query the Web of Linked Data

Live in your browser, powered by Comunica.





Linked Data Fragments

Choose datasources:

België gewesten ✕
België provincies ✕

Type or pick a query:

 SPARQL
  GraphQL-LD

```

1 SELECT ?f
2 WHERE {
3   gewest:Vlaanderen my:hasExactGeometry ?aGeom .
4   ?aGeom geo:asWKT ?aWKT .
5   ?f a my:Provincie .
6   ?f my:hasExactGeometry ?fGeom .
7   ?fGeom geo:asWKT ?fWKT .
8   FILTER (geof:sfContains(?aWKT, ?fWKT))
9 }


```


Execute query


5 results in 0.8s


Query results:

 <http://example.org/provincie#WestVlaanderen>

 <http://example.org/provincie#OostVlaanderen>

 <http://example.org/provincie#Antwerpen>

 <http://example.org/provincie#VlaamsBrabant>

 <http://example.org/provincie#Limburg>

Execution log:

Requesting <https://gist.githubusercontent.com/dreeki/e48bbe533a4b1191045b3652ff2c9c81/raw/ca8f5ed2856a2334>

Requesting <https://gist.githubusercontent.com/dreeki/e48bbe533a4b1191045b3652ff2c9c81/raw/ca8f5ed2856a2334>

Identified as file source: <https://gist.githubusercontent.com/dreeki/e48bbe533a4b1191045b3652ff2c9c81/raw/ca8f5ed2856a2334>

Identified as file source: <https://gist.githubusercontent.com/dreeki/e48bbe533a4b1191045b3652ff2c9c81/raw/ca8f5ed2856a2334>

Discover how **Linked Data Fragments** enable Web-scale querying of Linked Data.

©2013–2019 **Ghent University** – imec, Belgium. [Source code](#).

Figuur 3.6: Screenshot van online geplaatste testomgeving.

3.7 Overzicht

Om kort een overzicht te schetsen, zal deze sectie de implementatie overlopen en hierbij beschrijven wat gemaakt is en wat nog te doen is. Bij de vorige secties werden de functionaliteiten besproken in volgorde van implementatie. In dit overzicht zullen ze besproken worden in volgorde van uitwerking. Er kan alleszinds wel besloten worden dat een complete implementatie van GeoSPARQL zeer veel werk vraagt. In het kader van deze masterproef is hier al een deel van gemaakt, maar hier moet nog verder aan gewerkt worden.

3.7.1 Huidige status

Bij deze implementatie wordt eerst gecontroleerd of het gaat over een GeoSPARQL functionaliteit. Wanneer dit het geval is, zal eerst de linker- en rechterparameter recursief opgelost worden, zodat de functionaliteit effectief opgelost kan worden. Bij het oplossen van deze functionaliteit zal eerst de “WKT string” omgezet worden naar een GeoJSON object, waarbij gecontroleerd wordt of er een projectie (= referentiestelsel) meegegeven is (indien niet wordt gewerkt met de standaard waarde, genaamd “WGS84”). Indien beide parameters van de functie niet dezelfde projectie zouden hebben, dan worden de coördinaten van de tweede parameter omgerekend naar de respectievelijke waarden van de projectie van de eerste parameter. Eenmaal dat dit gedaan is kan de functie eenvoudig uitgewerkt worden met behulp van “Turf.js”. Dit resultaat wordt dan teruggegeven, zodat de boomstructuur dit kan gebruiken voor de verdere uitwerking van de recursieve structuur. Voor niet-topologische functies kan het nodig zijn om een GeoJSON object terug te geven. Om een uniforme werking van sparql te kunnen behouden, wordt dit GeoJSON object opnieuw geserialiseerd naar WKT formaat.

3.7.2 Toekomstwerk

Aangezien deze implementatie slechts een beperkte implementatie van GeoSPARQL is, zal hier in de toekomst nog verder aan gewerkt moeten worden. Er wordt momenteel slechts in zeer beperkte mate ondersteuning aangeboden voor het gebruik van “Feature”. Er is ook enkel ondersteuning voor “WKT strings” en niet voor het alternatieve “GML” formaat. Vervolgens moet er een aanvulling gemaakt worden van zowel de topologische functies als de niet-topologische functies. Bij de topologische functies is enkel gekeken naar de “Simple Features” familie (en deze is ook niet helemaal compleet), maar er zijn nog twee andere families. Bij de niet-topologische functies zijn er slechts twee functies geïmplementeerd, zijnde de “union” en de “intersection” functies. Hier zijn er nog verscheidene andere functies. Ten slotte moet het mogelijk zijn om deze functies te gebruiken in de vorm van predicaat. Hiervoor moet er een functionaliteit zijn

voor het herschrijven van de queries. Deze is totaal niet gebruikt, maar is wel nodig voor een volledige implementatie van GeoSPARQL.

“I alone cannot change the world, but I can cast a stone across the water to create many ripples.”

~Mother Teresa

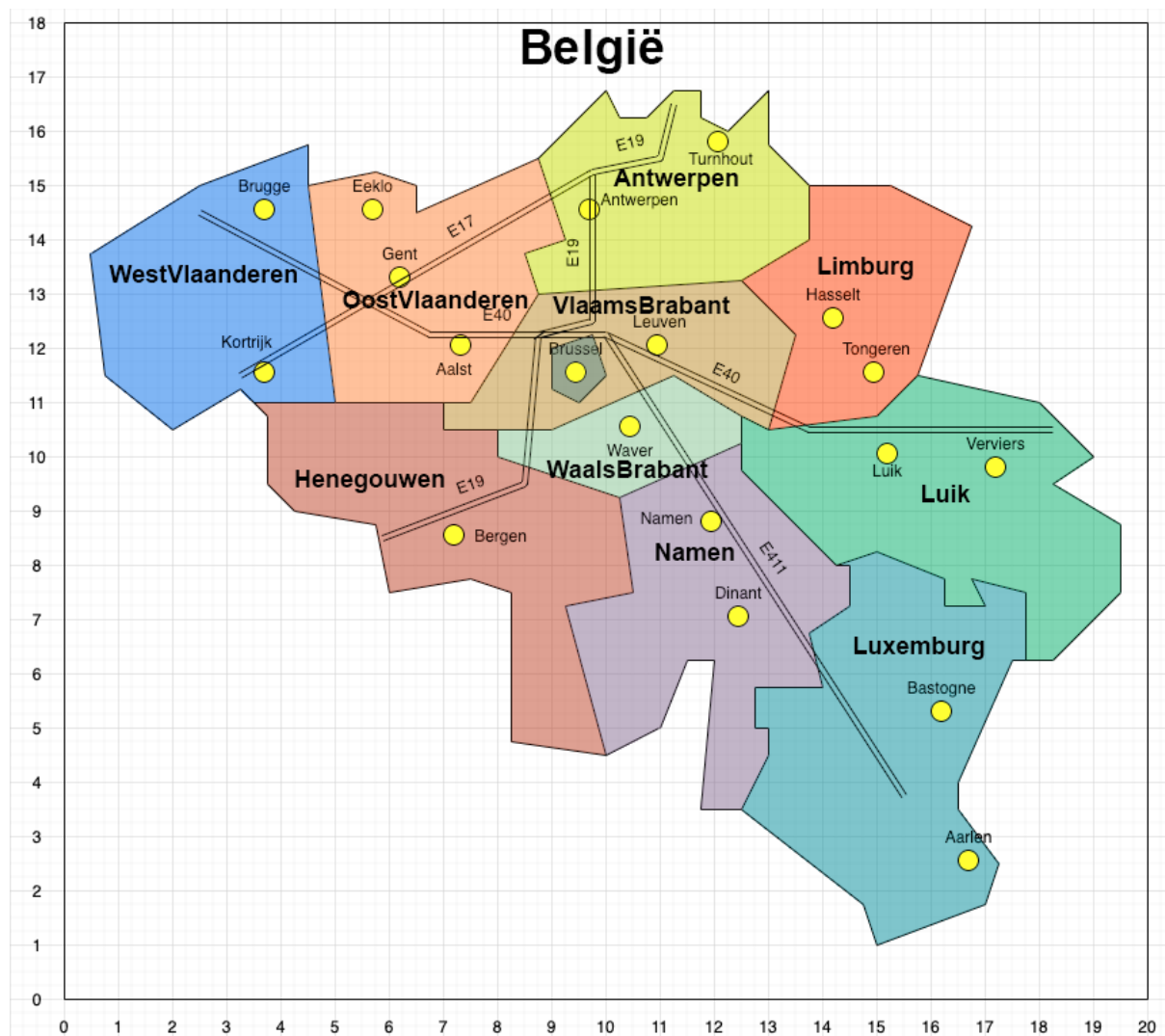
4

Interfaces

Het doel van deze masterproef is het maken van een implementatie van GeoSPARQL in Comunica en te controleren of het hierbij mogelijk is om GeoSPARQL functionaliteiten te implementeren in deze *query engine* die over verschillende heterogene data bronnen kan queryen. Hierbij worden de gegevens op de gebruikelijke manier opgehaald, maar worden de geospatiale relaties uitgerekend op de client. Hierbij moet gecontroleerd worden of dit mogelijk is bij de bronnen die momenteel door Comunica ondersteund worden. De belangrijkste bronnen zijn “data dumps”, “triple pattern fragment interfaces” en “sparql endpoints”. In de komende secties wordt besproken of dit mogelijk is en hoe dit werkt. De eerstvolgende sectie geeft echter een kort overzicht van de gebruikte dataset voor deze tests.

4.1 Testset

Bij de testset is ervoor gekozen om een oppervlakkige tekening te maken van België in een aangepaste schaal. Deze keuze is gemaakt om meerdere redenen. Om te beginnen laat deze aangepaste schaal zeer gemakkelijk toe om (als mens) vlakken te beschrijven. Aangezien dit een technisch en bovendien vooruitstrevend onderwerp is, is het handig om terug te keren naar een bekender terrein. Daarom is de bekendheid van deze *use case* meteen de tweede reden van deze keuze. Een derde reden is dat deze dataset eerder klein is, waardoor mogelijke fouten of



Figuur 4.1: Testset voor het testen van de verschillende bronnen.

onlogische oplossingen hierbij nogmaals getest worden. Ten slotte is dit ook de perfecte dataset voor het geven van demonstraties omdat deze set herkenbaar is voor het publiek. Een visualisatie van deze dataset is te zien in Figuur 4.1. De effectieve dataset is dan weer te vinden op GitHub Gist¹. Deze dataset is bovendien opgesplitst in vijf verschillende bestanden (namelijk: “land”, “gewest”, “provincie”, “weg” en “stad”) zodat deze dataset bruikbaar is voor het verifiëren of gefedereerd queryen nog steeds mogelijk is. Het geheel van deze testing wordt uitgevoerd in de testomgeving die besproken werd in Sectie 3.6. Hierbij zorgt de *execution log* ervoor dat alles duidelijk is hoe het geheel in zijn werk gaat.

¹<https://gist.github.com/dreeki/e48bbe533a4b1191045b3652ff2c9c81>

4.1.1 Queries

Voor de uitvoering van de tests zijn verschillende queries opgesteld om mee te testen. In de testomgeving kunnen hierbij vervolgens de bronnen aangepast worden naar de te testen bronnen. Echter moet opgemerkt worden dat de ontologieën bij de testomgeving reeds in code aangegeven werden. Bijgevolg worden deze niet opnieuw mee opgenomen in de geschreven query, maar in de achtergrond worden deze dus wel nog gebruikt.

Query 1

De eerste *query* zoekt naar de provincies die binnen Vlaanderen liggen. Om dit te kunnen doen zal de *query engine* eerst de vorm van Vlaanderen zelf opzoeken. Daarna zal hij de vorm van alle provincies zoeken zodat hij met de “sfContains” functie van GeoSPARQL uiteindelijk kan filteren. Deze query is te zien in Codefragment 4.1.

```
SELECT ?f
WHERE {
  gewest:Vlaanderen my:hasExactGeometry ?aGeom .
  ?aGeom geo:asWKT ?aWKT .
  ?f a my:Provincie .
  ?f my:hasExactGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .
  FILTER (geof:sfContains(?aWKT, ?fWKT))
}
```

Codefragment 4.1: Query die alle provincies in Vlaanderen zoekt.

Query 2

De tweede *query* is zeer gelijkaardig aan de eerste, maar hier is één groot verschil. Deze *query* zoekt naar alle objecten die binnen Vlaanderen ligt. Aangezien “sfContains” functie stelt dat een geospaatial identieke vorm steeds binnen de andere ligt, zou deze query Vlaanderen zelf ook als een oplossing zien. Aangezien dit niet het verwachte resultaat is, wordt gebruik gemaakt van de negatie van de “sameterm” functie van SPARQL. Dit wijst er nogmaals op dat een werkende implementatie van SPARQL een vereiste is voor het maken van een GeoSPARQL implementatie. Deze query is te zien in Codefragment 4.2.

Query 3

De derde query zoekt dan weer de wegen en provincies die binnen België liggen. Deze query toont nogmaals aan hoe gelijkaardig SQL en SPARQL zijn. Deze query is te zien in Codefragment 4.3.

```

SELECT ?f
WHERE {
  gewest:Vlaanderen my:hasExactGeometry ?aGeom .
  ?aGeom geo:asWKT ?aWKT .
  ?f my:hasExactGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .
  FILTER (geof:sfContains(?aWKT, ?fWKT) && !sameterm(?aWKT, ?fWKT))
}

```

Codefragment 4.2: Query die geospatiale objecten in Vlaanderen zoekt.

```

SELECT ?f
WHERE {
  land:België my:hasExactGeometry ?aGeom .
  ?aGeom geo:asWKT ?aWKT .
  {
    ?f a my:Provincie .
  }
  UNION
  {
    ?f a my:Weg .
  }
  ?f my:hasExactGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .
  FILTER (geof:sfContains(?aWKT, ?fWKT))
}

```

Codefragment 4.3: Query die provincies en wegen in België zoekt.

Query 4

Bij de vierde query worden alle wegen die door Oost-Vlaanderen gaan opgezocht. Dit zou bijvoorbeeld handig kunnen zijn wanneer iemand de snelwegen wil vinden die eenvoudig bereikbaar zijn vanuit Oost-Vlaanderen. Hierbij wordt de functie “sfIntersects” van GeoSPARQL gebruikt. Deze query is te zien in Codefragment 4.4.

```

SELECT ?f
WHERE {
  prov:OostVlaanderen my:hasExactGeometry ?aGeom .
  ?aGeom geo:asWKT ?aWKT .
  ?f a my:Weg .
  ?f my:hasExactGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .
  FILTER (geof:sfIntersects(?aWKT, ?fWKT))
}

```

Codefragment 4.4: Query die wegen die door Oost-Vlaanderen lopen, zoekt.

Query 5

De vijfde query toont aan dat het mogelijk is om manueel een vorm te voorzien om op te filteren. Deze vorm staat (bij de aangepaste schaal) voor de *bounding box* van Vlaams-Brabant en Waals-Brabant. Deze query zal alles weergeven dat zich binnen deze vorm bevindt. Om te variëren op vlak van gebruikte functie, wordt hier de “sfWithin” functie van GeoSPARQL gebruikt. Dit zou echter evengoed mogelijk zijn met de “sfContains” functie. Deze query is te zien in Codefragment 4.5.

```
SELECT ?f
WHERE {
  ?f my:hasExactGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .
  FILTER (geof:sfWithin(?fWKT, '''Polygon((7 9.25, 13.5 9.25, 13.5 13.25,
  7 13.25, 7 9.25))'''^geo:wktLiteral))
}
```

Codefragment 4.5: Query geospatiale objecten binnen de *bounding box* van Brabant zoekt.

4.2 Data dump

De eerste bron om te controleren wordt gebruikt als baseline. Dit is de “data dump”. Dit is een gewoon bestand in RDF formaat dat door de *query engine* opgehaald (lees gedownload) wordt. Vervolgens wordt door de *query engine* gecontroleerd of het effectief wel een bestandsbron is. Eenmaal dit voldaan is, worden steeds de kleinste patronen gezocht die voldoen aan de bron. Dit is nodig omdat de *query engine* zo op de meest performante manier de *joins* kan uitvoeren. Zo kan ten slotte de filter-functie de overbodige oplossingen weghalen. Deze filter-functie is voorzien door GeoSPARQL. Bovendien voorziet Comunica functionaliteiten om data-entiteiten uit de databron te extraheren, wat hier gebeurt op de client. Over deze entiteiten moet een filter functie geëvalueerd kunnen worden. Hierdoor is het mogelijk om data dumps te queryen met GeoSPARQL. Aangezien data dumps letterlijk bestanden zijn zonder een eigen voorziening van logica, is het triviaal dat dit afgehandeld moet kunnen worden. De data dump wordt daarom de *baseline* van dit onderzoek, waarbij er gepoogd wordt om dezelfde resultaten bij andere bronnen ook te behalen.

Bij het testen van de data dump worden de GitHub Gist bestanden (zie Sectie 4.1) rechtstreeks gebruikt. Hier is geen enkel ander programma dat als aanspreekpunt gebruikt wordt. Hierbij is dus ook duidelijk dat het beschreven proces van hierboven correct doorlopen is. Dit is bovendien de manier van werken die gebruikt is bij het maken en controleren van de implementatie.

4.3 Triple pattern fragment interface

De volgende bron is de “triple pattern fragment interface”. Deze server staat tussen de op te vragen bestanden (dus de effectieve gegevens) en de client. Deze server zal ervoor zorgen dat het niet langer nodig is om alle data op te halen, maar in plaats daarvan zal de server de SPARQL query splitsen in verschillende triple pattern fragment requests. Deze vragen alle triples die voldoen aan een enkel tripple pattern fragment en *joinen* dan de resultaten op de client. De filtering gebeurt daarna ook op de client.

Net zoals bij de data dump (zie Sectie 4.2) vraagt de *query engine* de bron op. Hierbij zal hij de bron identificeren als een “qpf source”, wat staat voor “Quad pattern fragment”. Dit is eigenlijk de *triple pattern fragment* met hierbij een extra veld (graph) toegevoegd, maar hier wordt niet verder op in gegaan. Vervolgens begint de *query engine* de query op te splitsen in *Triple Pattern Fragment* (TPF)-queries, zodat deze TPF-queries geoptimaliseerd kunnen worden in een volgorde die gebaseerd is op de initiële count query. Hierna worden deze één voor één uitgevoerd. Hij zal vervolgens de kleinste patronen opvragen, zodat de correcte informatie opgevraagd kan worden, met hierbij een minimale hoeveelheid aan overbodige informatie. Dit is enkel mogelijk dankzij de TPF-interface. Dankzij deze manier van werken kan wederom de uiteindelijke filtering van de *queries* louter op de client gebeuren.

Voor het opzetten van deze test is gebruik gemaakt van de “Linked Data Fragments Server”. Deze bouwt een TPF-interface op boven een set van bronbestanden, waarvoor opnieuw de bestanden van GitHub Gist gebruikt zijn, net zoals bij de data dump. Op deze manier is ook verzekerd dat er met dezelfde gegevens gewerkt wordt.

Als kleine opmerking kan nog vermeld worden dat een TPF-interface gebruikt wordt om twee redenen. Ten eerste hoeft de client zo niet alle data te downloaden, maar kan de server slechts een fragment (vandaar de naam, deze komt eigenlijk van “Linked Data Fragments”) van deze data teruggeven. De tweede reden is dat deze filtering meestal (= niet in alle gevallen) zorgt voor een verbeterde performantie. Bij het testen was dit echter niet terug te vinden. Zo blijkt het uitvoeren van de queries met de data dump sneller te gaan dan met de TPF-interface. Hier is echter een logische verklaring voor. De datasets die gebruikt zijn, zijn relatief kleine datasets. Bovendien bevatten deze enkel de noodzakelijke gegevens, waardoor de dataset volledig nodig is voor het uitvoeren van de query. Hierdoor kan er niet genoten worden van de voordelen van de TPF-interface, maar wordt enkel de extra *overhead* waargenomen. Dit gaat echter buiten de *scope* van deze masterproef, daarom werd hier verder geen onderzoek naar gedaan, noch benchmarking van de performantie. Dit is eerder een opmerking bij de ondervindingen.

4.4 SPARQL endpoint

De laatste bron om te testen is meteen de moeilijkste. Bij het SPARQL *endpoint* is het de bedoeling dat een GeoSPARQL query uitgevoerd kan worden door de gegevens op te vragen aan dit SPARQL *endpoint*. Comunica zelf is gemaakt om queries uit te voeren over RDF-bronnen, wat het zeer handig maakt om SPARQL queries uit te voeren. Hierdoor lijkt het logisch om de query in zijn geheel door te sturen in het geval van een SPARQL *endpoint* als bron. Dit SPARQL *endpoint* is namelijk in staat om volledig automoot een antwoord te geven op de query. Dit is echter niet hoe het in zijn werk gaat. Eén van de redenen hiervoor is dat gefedereerd queryen niet mogelijk zou zijn wanneer er naast het SPARQL *endpoint* nog een andere bron zou zijn. Zo moet het samenvoegen van de antwoorden op de client gebeuren.

Bij een SPARQL *endpoint* zal de *query engine* de verschillende RDF-triples van de query overlopen. Dit doet hij in twee stappen. De eerste stap is een “count”, zodat hij weet hoeveel RDF-triples van de bron overeen komen met de RDF-triples van de query. Zo weet de *query engine* welke volgorde optimaal is om de data op te halen, zodat hij dit optimaal kan joinen. De tweede stap is het effectieve ophalen van het resultaat, waarbij hij dus alle antwoorden vraagt die voldoen aan slechts één RDF-triple van de query. Wanneer dit voor alles gedaan is, zoekt hij het kleinste patroon, zodat hij vervolgens de matchende RDF-triples kan ophalen. Het kleinste patroon wordt gekozen om het aantal matchende resultaten te minimaliseren voor performantie redenen.

Op deze wijze wordt uiteindelijk alle benodigde informatie uit het SPARQL *endpoint* systematisch opgehaald, zodat de filter bij de oorspronkelijke query onafhankelijk van de bronnen kan uitgevoerd worden. Dit betekent dat de filterfuncties steeds dezelfde implementatie hebben (namelijk deze van de *query engine* op de client, niet deze van de bron). Dit laat toe om te filteren met de GeoSPARQL functies. Dankzij deze werkwijze is het effectief mogelijk om de GeoSPARQL functionaliteit toe te passen bij het opvragen aan een SPARQL *endpoint*.

“Dream big and dare to fail.”

~Norman Vaughan

5

Conclusie

In dit hoofdstuk worden de resultaten, besproken in Hoofdstuk 4, geïnterpreteerd. Er wordt een antwoord geformuleerd op de vragen uit Sectie 1.3.

Onderzoeksvraag: Welke “Linked Data publicatie”-interfaces kunnen uitgebreid worden met GeoSPARQL-functionaliteiten door de filtering op de client uit te voeren?

De masterproef brengt een oplossing om de “Linked Data publicatie”-interfaces uit te breiden met GeoSPARQL-functionaliteiten. Dit betekent dat er gelinkte data online gepubliceerd worden. Deze data kunnen opgehaald worden met behulp van SPARQL (zie Sectie 2.4).

De vraag is hoe deze opvraging kan uitgebreid worden met GeoSPARQL-functionaliteiten. Hiervoor wordt gebruik gemaakt van de al bestaande implementatie van Comunica. Door Comunica uit te breiden met deze GeoSPARQL-functionaliteiten wordt gepoogd om op dezelfde manier te kunnen werken als voordien, maar ditmaal met GeoSPARQL-functionaliteiten. Specifiek hierbij worden de data opgehaald en gefilterd op de client zelf (zoals besproken in Hoofdstuk 4).

De meest voorkomende “Linked Data publicatie”-interfaces voor het gebruik van geospatiale data zijn “data dumps”, “TPF interfaces” en “SPARQL endpoints”.

Hypothese 1: Het is mogelijk om GeoSPARQL queries uit te voeren over “data

dumps” waarbij de filtering op de client-side gebeurt.

Bij een “data dump” worden de data volledig gedownload op de client. Hier zal de client vervolgens de resultaten joinen zoals nodig in de query. Ten slotte zal de client over de volledige dataset filteren, om zo tot het correcte resultaat te komen.

Hieruit volgt dat Hypothese 1 correct is.

Hypothese 2: Het is mogelijk om GeoSPARQL queries uit te voeren over “TPF interfaces” door de filtering op de client-side uit te voeren.

De “TPF interface” is een server die een dataset bevat. Deze server bevat functionaliteiten om te kunnen antwoorden op vragen naar een *triple pattern fragment*. Zo wordt de volledige query opgesplitst, zodat de “TPF interface” het kleinst mogelijke deel van de dataset (dat alle vereiste data bevat) kan terug geven. Hierbij zal de client wederom de resultaten joinen om hierop te kunnen filteren zodat het correcte resultaat bekomen kan worden.

Hieruit volgt dat ook Hypothese 2 correct is.

Hypothese 3: Het uitvoeren van GeoSPARQL queries op een “SPARQL endpoint” is niet vanzelfsprekend. Het is echter mogelijk door de filtering op de client-side uit te voeren.

Een “SPARQL endpoint” is zelfstandig in staat om te antwoorden op een SPARQL query. Hierbij is er geen optie om een GeoSPARQL query door te sturen omdat het “SPARQL endpoint” hier niet kan op antwoorden. Bij een “SPARQL endpoint” wordt dit probleem aangepakt door niet de volledige query door te sturen, maar in de plaats te tellen hoeveel antwoorden er zijn op elk *triple pattern fragment* in de query. Zo kan de client zelf beslissen welk fragment nodig is om het kleinst mogelijke patroon te vinden. Hierdoor kan het joinen van het resultaat op de client gebeuren. Ook hierbij is dus de laatste stap om het resultaat te filteren op de client.

Hieruit volgt dat ook Hypothese 3 correct is.

5.1 Toekomstig werk

De gemaakte implementatie is slechts een beperkte implementatie van GeoSPARQL (zoals vermeld in Subsectie 3.7.2). Bij verdere implementatie hiervan moet gecontroleerd worden in hoeverre de libraries (zoals “Turf” en “Proj4”) de vereiste functionaliteiten correct ondersteunen. Enerzijds voorziet Turf een groot arsenaal aan geospatiale functionaliteiten, maar wanneer deze niet helemaal kloppen met de verwachtingen is het niet mogelijk om deze manueel aan te passen, om andere conclusies te trekken. Hierbij zou het een meerwaarde zijn om een *library* te maken

die eerder werkt op basis van de DE-9IM intersectie. Hiermee wordt bedoeld dat de DE-9IM intersectie aangegeven zou worden door deze nieuwe *library*. Aan de hand van DE-9IM worden alle vereisten van de GeoSPARQL-functionaliteiten uitgedrukt. Zo zou het eenvoudiger zijn om een specifieke functionaliteit van GeoSPARQL correct te implementeren.

Verder zou het ideaal zijn, moest Sparqlee uitgebreid worden met *custom* functies, zodat de GeoSPARQL-functionaliteiten in Comunica zelf geïmplementeerd kunnen worden. Deze zouden dan geïnjecteerd moeten worden in Sparqlee. Op deze manier kan de modulariteit van Comunica volledig benut worden.

Als laatste blijft het steeds een zoektocht naar de meest performante manier om alle data te verwerken. In deze masterproef werd rekening gehouden met performantie, maar aangezien dit niet de focus was, is hier niet te diep op ingegaan. Zo dient er een benchmarking te gebeuren ter controle of de gemaakte oplossingen schaalbaar zijn.

5.2 Tot slot

In deze masterproef is een implementatie gemaakt van GeoSPARQL. Zoals besproken in Hoofdstuk 3 is deze implementatie grotendeels gemaakt in Sparqlee, met een eigen GeoSPARQL actor in Comunica. Hierbij is gebruik gemaakt van de libraries “Turf”, “Proj4” en “Terraformer”. Bij deze implementatie is bovendien een client gemaakt, zodat dit geheel in een visuele omgeving zichtbaar is. Deze omgeving voorziet voldoende logs om te controleren hoe alles samenwerkt. Daarnaast is het mogelijk om de performantie te controleren, aangezien deze client meegeeft hoelang de query duurde om uit te voeren.

Vervolgens werd deze implementatie toegepast in Hoofdstuk 4. Hierbij werd getest of de verschillende interfaces uitgebreid konden worden met GeoSPARQL-functionaliteiten. In dit hoofdstuk wordt beschreven hoe het geheel in zijn werk gaat. Aan de hand van deze beschrijving wordt nogmaals duidelijk waarom het filteren op de client belangrijk is.

In Hoofdstuk 5 worden deze resultaten opnieuw geïnterpreteerd. Zo kan een concreet antwoord gevormd worden op de hypothesen die gesteld zijn in Sectie 1.3. Tot slot kan geconcludeerd worden dat deze hypothesen correct zijn. Dit betekent dat zowel “data dumps”, als “TPF interfaces”, als “SPARQL endpoints” uitgebreid kunnen worden met GeoSPARQL-functionaliteiten door de filtering op de client uit te voeren.

Bibliografie

- [1] T. Berners-Lee, J. Hendler, and O. Lassila, “The semantic web,” *Scientific american*, vol. 284, no. 5, pp. 34–43, 2001.
- [2] I. Horrocks, B. Parsia, P. Patel-Schneider, and J. Hendler, “Semantic web architecture: Stack or two towers?” in *International Workshop on Principles and Practice of Semantic Web Reasoning*. Springer, 2005, pp. 37–41.
- [3] T. Berners-Lee, “Www past, present and future,” 2005. [Online]. Available: <https://www.w3.org/2003/Talks/0922-rsoc-tbl/>
- [4] *Unicode characters — A Global Standard to Support ALL the World’s Languages*.
- [5] *URI-spec*.
- [6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau *et al.*, “Extensible markup language (xml) 1.0,” 2000.
- [7] T. Bray, D. Hollander, A. Layman, and R. Tobin, “Namespaces in xml,” *World Wide Web Consortium Recommendation REC-xml-names-19990114*. <http://www.w3.org/TR/1999/REC-xml-names-19990114>, 1999.
- [8] T. Gruber, “What is an ontology?” 2018.
- [9] T. Berners Lee, “Linked data,” 2006.
- [10] O. Lassila, R. R. Swick *et al.*, “Resource description framework (rdf) model and syntax specification,” 1998.
- [11] G. Klyne, J. J. Carroll, and B. McBride, “Resource description framework (rdf): concepts and abstract syntax, 2004,” *February*. URL: <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210>, 2009.
- [12] T. Heath and C. Bizer, “Linked data: Evolving the web into a global data space,” *Synthesis lectures on the semantic web: theory and technology*, vol. 1, no. 1, pp. 1–136, 2011.

- [13] F. Manola, E. Miller, B. McBride *et al.*, “Rdf primer,” *W3C recommendation*, vol. 10, no. 1-107, p. 6, 2004.
- [14] B. Adida, I. Herman, M. Sporny, and M. Birbeck, “Rdfa 1.1 primer,” *Rich Structured Data Markup for Web Documents*. Online verfügbar unter <http://www.w3.org/TR/xhtml-rdfa-primer/>, zuletzt geprüft am, vol. 9, p. 2016, 2012.
- [15] D. Beckett, T. Berners-Lee, E. Prud’hommeaux, and G. Carothers, “Rdf 1.1 turtle,” *World Wide Web Consortium*, 2014.
- [16] D. Beckett and J. Grant, “Rdf 1.1 n-triples,” URL: <https://www.w3.org/TR/n-triples>, 2014.
- [17] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and M. Birbeck, “Json-ld syntax 1.0,” *W3C Community Group Final Specification*, 2012.
- [18] D. Beckett, “Re: What does sparql stand for?” mail, 2011, uRL: <https://lists.w3.org/Archives/Public/semantic-web/2011Oct/0041>.
- [19] S. Harris and A. Seaborne, “Sparql 1.1 query language,” *World Wide Web Consortium*, 2013.
- [20] R. Taelman, J. Van Herwegen, M. Vander Sande, and R. Verborgh, “Comunica: a modular sparql query engine for the web,” in *International Semantic Web Conference*. Springer, 2018, pp. 239–255.
- [21] “Open geospatial consortium,” uRL: <https://ogc.org>.
- [22] “Geosparql support: What is geosparql,” uRL: <http://graphdb.ontotext.com/documentation/standard/geosparql-support.html>.
- [23] J. Shen, M. Chen, and X. Liu, “Classification of topological relations between spatial objects in two-dimensional space within the dimensionally extended 9-intersection model,” *Transactions in GIS*, vol. 22, no. 2, pp. 514–541, 2018.