Vrije Universiteit Brussel

Faculteit Wetenschappen,
Departement Informatica
Web & Information System Engineering (WISE)

Query Enrichment op het Semantisch Web

Een meerwaarde voor ontwikkelaar en eindgebruiker

Proefschrift ingediend met het oog op het behalen van de graad van
Licenciaat Toegepaste Informatica door

Jurgen David

Academiejaar 2006-2007

Promotor: Prof. Dr. Ir. Geert-Jan Houben
Begeleidende Assistent: Dr. Sven Casteleyn

Samenvatting

De interesse in het Semantisch Web neemt steeds toe. Men begint het potentieel van semantisch geannoteerde data vaker in te zien. Een mooie toepassing van het Semantisch Web is een nieuwe, semantische laag boven reeds bestaande applicaties te plaatsen om de voordelen van metadata uit te buiten. Dit zonder de reeds bestaande applicaties al te veel moeten veranderen. Als men ook nog eens rekening houdt met relaties tussen verschillende databronnen, kan men zeer ver gaan in het ontwikkelen van nieuwe applicaties.

Wat in dit document voorgesteld wordt, is een aanpak om SeRQL-queries gedistribueerd uit te voeren zonder volledige repositories lokaal te kopiëren. Het systeem dat uiteindelijk ontwikkeld werd, werd tot "DiSemDa" gedoopt. DiSemDa is een framework dat toelaat verschillende bronnen RDF-data met elkaar te verbinden. De input, een SeRQL-query, wordt opgesplitst in verschillende subqueries. Sesame[9] wordt gebruik om dergelijke subqueries uit te voeren op een server. Tijdens de uitvoering wordt steeds rekening gehouden met relaties tussen databronnen. Vervolgens worden alle deelresultaten samengevoegd tot 1 geheel en teruggegeven naar de gebruiker toe.

Deze aanpak heeft enkele belangrijke troeven. Ten eerste kunnen relaties tussen verschillende bronnen data uitgebuit worden. DiSemDa is ook heel dynamisch, er kunnen namelijk steeds nieuwe bronnen aan toegevoegd worden. Dit alles gebeurt met minimale menselijke interactie.

Overzicht

De eerste sectie van dit document is een inleidend hoofdstuk dat de het probleem en de oplossing schetst. "DiSemDa" wordt volledig uit de doeken gedaan in het tweede hoofdstuk. De sectie die hierop volgt, bespreekt enkele eigenschappen en toepassingen van "DiSemDa". Tot slot bevat de laatste sectie een conclusie en een overzicht van mogelijke uitbreidingen.

Inhoud

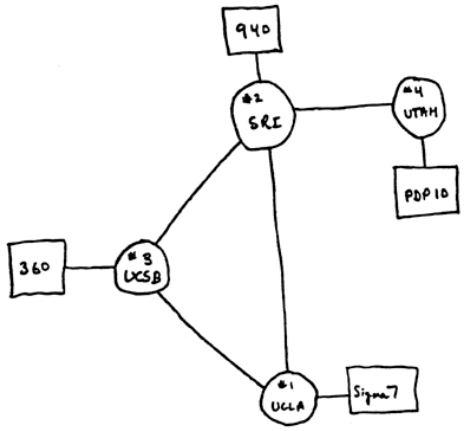
1 Inleiding

De computertechnologie is in de loop der jaren erg veranderd, alsook het gebruik van een computer. Er worden steeds meer gesofisticeerde programma's geschreven. De komst van het Internet heeft dit een flinke duw in de rug gegeven: vele programma's verlenen informatie via het Web. Het Semantisch Web maakt zijn opmars. De bestaande technologie wordt op een andere manier aangewend. Zo wordt nog steeds het bestaande Internet gebruikt. Er zijn natuurlijk wat extra snufjes: er wordt gebruik gemaakt van met metadata verrijkte data, ook wel semantisch geannoteerde data genoemd. Dit laat programma's toe te redeneren over data zelf. De toepassingen hiervan kunnen heel wat verschillende richtingen uitgaan.

Verder worden er in dit hoofdstuk "mapping frameworks" besproken. Deze frameworks proberen relaties tussen verschillende bronnen semantisch geannoteerde data te leggen. Er zal aangetoond worden dat de oplossing die deze frameworks aanbieden, niet voldoen voor de probleemstelling aangehaald in dit document. Er zal ook aangegeven worden op welke manier de aanpak uitgeschreven in dit document, verschilt met andere theorieën die terug te vinden zijn in de literatuur. De doelstellingen voor de oplossing worden duidelijk beschreven. Tot slot wordt uitgelegd hoe de voorgestelde oplossing opgebouwd is en welke voordelen het biedt.

1.1 Historisch Overzicht

In 1969 werd met ARPANET de eerste stap gezet naar een wereldwijd computernetwerk. Dit netwerk was een aaneenschakeling van lokale netwerken om onderzoeksbijdragen uit te wisselen tussen University of California Los Angeles (UCLA), Stanford Research Institute (SRI), University of California Santa Barbara (UCSB) en University of Utah (figuur 1). Het groeide steeds aan en werd internationaler. Wanneer er in 1983 werd overgeschakeld op het TCP/IP-protocol, werd het Internet in zijn huidige vorm geboren. Maar het was pas in de jaren '90 dat het grote publiek de weg naar het wereldwijde Web had gevonden.



Figuur 1: Het vroege ARPANET [1]

De komst van het Internet heeft er voor gezorgd dat data efficiënt uitgewisseld kon worden. De meest primitieve vorm is eigenlijk de vorm die nog het meest gebruikt wordt: een website. Er wordt data tentoongesteld aan de wijde wereld zonder meer. Nu komen er steeds meer dynamische websites waar gebruikers zelf informatie kunnen ingeven, opzoeken, ... etc. Er wordt vaak gebruik gemaakt van een onderliggende database.

Maar de komst van het Internet heeft er ook toe geleid dat er gedistribueerde systemen ontstaan zijn. Men kan bijvoorbeeld instructies op een server uitvoeren die aan de andere kant van de wereld staat. Zo staat ook data gedistribueerd op het Web, data die soms onderling gerelateerd is. Als er bovendien rekening gehouden wordt met het feit dat iedereen zijn eigen datamodel kan aanmaken, dan valt het niet te betwijfelen dat er problemen ontstaat om relaties tussen deze data voor te stellen. Deze verscheidenheid is ook net wat het Web zo interessant maakt, maar soms moeilijk om mee te werken.

Een stap in de goede richting was het idee van het "Semantisch Web". Het idee hierachter is dat er semantiek toegevoegd wordt aan de data die beschikbaar is op het wereldwijde Web. Deze toegevoegde data wordt "metadata" genoemd, het is data over data. Deze extra informatie laat toe dat programma's kunnen bepalen wat de betekenis is van data. Dit is al een goed begin, maar er is natuurlijk ook noodzaak om delen van deze data te kunnen selecteren. Voor dit doeleinde zijn query-talen ontwikkeld specifiek voor semantisch geannoteerde data zoals SeRQL[11] en SPARQL[12]

1.2 Achtergrond

Er zijn vandaag de dag toch al een aantal mogelijke tools en frameworks dat men kan gebruiken wanneer men een applicatie wil ontwikkelen die gebruik maakt van semantisch geannoteerde data. In deze sectie volgt een selectie van de interessantste projecten in het kader van dit onderzoek.

1.2.1 Verkrijgen van Semantisch Verrijkte Data

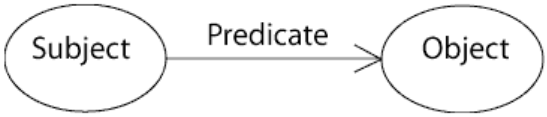
Het verkrijgen van semantisch verrijkte data is de eerste stap in de omschakeling naar het Semantisch Web, maar geen onbelangrijke. Hier wordt immers de basis gelegd voor de meerwaarde die later geboden kan worden. Er is reeds veel informatie beschikbaar in relationele databases. Het is dus ook voordelig om deze data te kunnen recupereren door er metadata aan toe te voegen. In [2] wordt aangegeven hoe informatie uit relationele databases kan omgezet worden naar data die geschikt is voor het Semantisch Web. Dit gebeurt in 2 stappen.

In de eerste stap wordt m.b.v "Relational.OWL" (zie [3]) informatie uit een relationele database gehaald en omgezet naar een eerder gedefinieerde ontologie. Dit model bevat echter nog geen semantiek, het is enkel een vertaling van de data uit de relationele database naar een ontologie, uitgedrukt in RDF[13]. De tweede stap in dat proces bestaat eruit om SPARQL te gebruiken om de "Relational.OWL"-ontologie om te vormen naar een doelontologie zodat men er echte semantiek kan aan toevoegen. Het voordeel tegenover sommige andere methoden is dat er gebruik kan gemaakt worden van een reeds bestaande query-taal en het dus niet nodig is een nieuwe query-taal onder de knie te krijgen. Maar er is dus nog steeds menselijke interactie nodig om een resultaat te bekomen dat betekenisvolle metadata bevat.

1.2.2 Resource Description Framework (RDF)

RDF[13] is een dataformaat dat voorgesteld kan worden als een RDF-graph. RDF-data wordt voorgesteld in triples van onderstaande componenten (figuur 2):

- *Subject*: het onderwerp waarover de data handelt
- *Predicate*: een predikaat kan gezien worden als een eigenschap van het subject in kwestie
- *Object*: een object is de waarde die gegeven wordt aan het predikaat



Figuur 2: RDF-graph [13]

RDF baseert zich eigenlijk op relaties tussen elementen om data voor te stellen. De subjects en objects zijn de nodes in een RDF-graph. Elk subject kan dus ook als object gebruikt worden en vice versa. Een node kan een URI[1] zijn, een verwijzing naar een andere node. Dit zorgt voor een grote flexibiliteit van RDF-data.

RDF is dus een graph, maar kan voorgesteld worden in verschillende formaten. Het meest gebruikte formaat is RDF/XML[15]: RDF-data wordt uitgedrukt in XML. Er zijn ook andere formaten beschikbaar zoals Turtle[16], Notation 3[17], N-Triples [18] en TRIX [19].

1.2.3 Sesame

Sesame[9] is een open-source framework voor RDF. Het is ontwikkeld in Java en ook beschikbaar als een Java-library. Sesame 2 heeft ondersteuning ingebouwd voor SeRQL[11] en SPARQL[12]. Via deze query-talen geeft Sesame de mogelijkheid om informatie op te vragen uit een repository. Het is niet mogelijk via dezelfe weg om data toe te voegen. Daarvoor is het nodig andere onderdelen van de Sesame API te gebruiken.

Sesame bevat ook een webapplicatie die als poort naar de onderliggende repositories fungeert. Om dit te laten werken is het wel vereist Apache Tomcat[20] webserver te gebruiken. Via een webclient, ook voorzien door de ontwikkelaars van Sesame, is het ook mogelijk informatie op te vragen met behulp van SeRQL of SPARQL. Deze webclient laat toe op een eenvoudige manier data toe te voegen aan repositories. Er worden heel wat RDF-formaten ondersteund: RDF/XML, Notation 3, N-Triples, Trix en Turtle. Ook biedt deze web-interface de mogelijkheid data te verwijderen of te navigeren door data.

Het scheiden van de web-client en de server heeft als voordeel dat men kan toelaten dat applicaties gebruik maken van Sesame terwijl niet iedereen op het net toegang heeft tot de data. Applicaties die gebruik maken van de Sesame library kunnen via een "HTTPRepository" een query uitvoeren op een server via het Internet. Dit is natuurlijk een groot voordeel voor gebruik op het Semantisch Web.

[1] URI: **U**nified **Resource I**dentifier

1.2.4 Mapping Frameworks

Eens er semantisch verrijkte data beschikbaar is, moeten er nog nuttige zaken mee kunnen gedaan worden. Het is de bedoeling een meerwaarde te bieden ten opzichte van traditionele systemen. De informatie op het Web zou veel rijker kunnen worden indien relaties tussen verschillende databronnen zouden kunnen gespecificeerd worden. Er zijn al een aantal "mapping frameworks" die een oplossing voor dit probleem trachten uit te werken. In [4] wordt er een overzicht gegeven van verschillende frameworks. Deze werden allemaal geëvalueerd op dezelfde criteria waarvan de resultaten in Figuur 3 terug te vinden zijn.

	MOMIS	LSD	CTXMATCH	GLUE	MAFRA	LOM	ONION	PROMPT	FCA-Merge
Input	Data model	Source schemas & their instances	Concepts in concept hierarchy	Two taxonomies with their data instances in ontologies	Two ontologies	Two lists of terms from two ontologies	Terms in two ontologies	Two input ontologies	Two input ontologies and a set of documents of concepts in ontologies
Output	An integrated global ontology (GVV)	pairs of related terms between a global and local schema	Semantic relation between concepts	A set of pairs of similar concepts	Mappings of two ontologies by the Semantic bridge ontology	A list of matched pairs of terms with score ranking similarity	Sets of Articulation rules between two ontologies	A merged ontology	A merged ontology
User interaction	The designer involves in schema annotation & sets a threshold for integration clusters for generating a GVV	The user provides mappings for training source & feedback on the proposed mappings.	No (CTXMATCH is an algorithm.)	User-defined mappings for training data , similarity measure, setting up the learner weight, and analyzing system's match suggestion	The domain expert interface with the similarity and semantic bridging modules and it has graphical user interface	It requires human validation at the end of the process.	A human expert chooses or deletes or modifies suggested matches using a GUI tools	The user accepts, Rejects , or adjusts system's suggestions.	Generating a merged ontology requires human interaction of the domain expert with background knowledge
Mapping strategy or algorithm	Name equality: Synonyms hyponyms Matching of clustering	Multi-strategy Learning approach : (machine Learning technique)	Logical deduction	Multi-strategy learning approach : (machine learning technique)	Semantic bridge	Lexical similarity whole term, word constituent, synset, and type matching	Linguistic matcher, Structure- , inference-based heuristics	Heuristic-based analyzer	Linguistic analysis & TITANIC algorithm for computation for pruned concept lattice
Structured knowledge	Yes	No	Yes	No	Yes	No	Yes	Yes	Yes
Instance-based knowledge	No	Yes	No	Yes	Yes	No	No	No	Yes
Lexical knowledge	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
domain knowledge	No	Yes	Yes	Yes	Yes	No	Yes	No	Yes

Figuur 3: Overzicht van mapping frameworks [4]

In datzelfde paper spreekt men over 3 verschillende categorieën waaronder men al deze tools kan onderbrengen. De eerste categorie is "ontology merging". Dit houdt in dat er 1 coherente ontologie aangemaakt wordt startende van 2 of meerdere bestaande ontologieën die data bevatten over hetzelfde domein. De tweede categorie "ontology integration" lijkt hier sterk op, het enige verschil is dat het over ontologieën gaat die betrekking hebben tot andere domeinen. De laatste categorie heet "ontology alignment" en is in het kader van dit document de meest interessante. Hier gaat het over het aanmaken van bruggen tussen 2 of meerdere ontologieën zodat deze consistent worden met elkaar. Er wordt geen nieuwe ontologie geproduceerd bij deze aanpak. Uit al deze informatie kan besloten worden dat het MAFRA[2] framework het kortst aansluit bij het thema in deze paper.

1.2.5 MAFRA

MAFRA[5] biedt een gedistribueerd mappingsproces aan. Het bestaat uit 5 horizontale modules (figuur 4):

1. *Lift and Normalization*: Deze module zorgt ervoor dat syntactische verschillen tussen verschillende databronnen weggewerkt worden door alle data te normaliseren naar een uniforme representatie, nl. RDF.

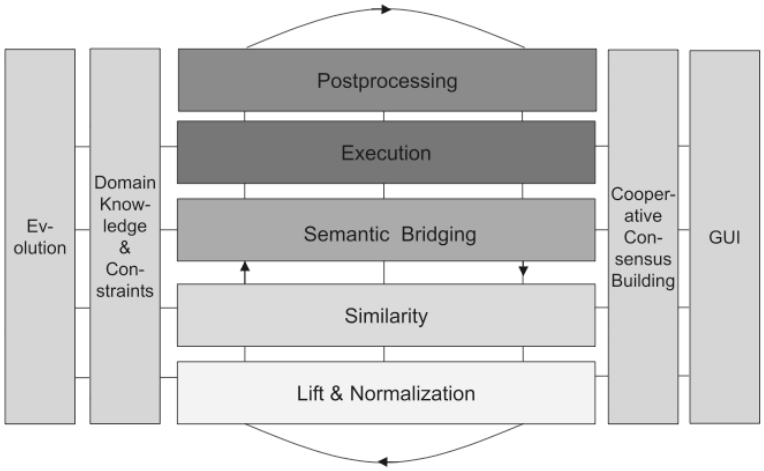
2. *Similarity*: MAFRA ondersteunt het zelfstandig vinden van mappings. In deze module wordt data nagekeken bijvoorbeeld op gelijkenis om zo mappings te kunnen opsporen. Er worden op verschillende niveau's naar gelijkenissen gezocht, onder andere op lexicale gelijkenissen, gelijke eigenschappen en dergelijke.

3. *Semantic Bridging*: Dit is in het kader van het onderzoek naar een gedistribueerd querysysteem waarschijnlijk wel het meest interessante onderdeel. De Semantic Bridge Ontology (SBO) is een heel expressieve ontologie om veel mogelijke relaties tussen concepten uit te drukken. Er wordt bijgehouden hoe een ontologie omgezet kan worden in een andere.

4. *Execution*: In deze module worden instanties van de doelontologie aangemaakt. Er wordt rekening gehouden met de Semantic Bridges.

5. *Post-processing*: Het resultaat van de vorige module wordt overlopen om te zien of de kwaliteit van de resultaten niet verbeterd kan worden.

[2] Ontology **Ma**pping **Fra**mework voor gedistribueerde ontologieën op het Semantisch Web.



Figuur 4: MAFRA conceptuele architectuur [5]

MAFRA is ook opgebouwd uit 4 verticale componenten die samenwerken met de horizontale modules tijdens het gehele proces:

1. *Evolution*: Deze module zorgt ervoor dat de Semantic Bridges tussen de ontologieën consistent blijven. Beide ontologiën veranderen immers.

2. *Cooperative Consensus Building*: Er zijn verschillende mogelijke mappings mogelijk tussen ontologieën. Deze module maakt het mogelijk te kiezen uit deze verschillende mogelijkheden. Enige menselijke interactie kan mogelijk zijn.

3. *Domain Constraints and Background Knowledge*: Door het toevoegen van achtergrondkennis van constraints van het domein waarin gewerkt wordt, kan de kwaliteit van de gelijkenissen die gevonden worden hoger liggen. Dit beïnvloedt de Semantic Bridges ook positief: ze zijn beter van kwaliteit en de minder goede Semantic Bridges kunnen geëlimineerd worden.

4. *Graphical User Interface*: Het mappingsproces op zich neemt veel tijd in beslag en er is een uitgebreide kennis van beide ontologiën nodig. Daarom kan een uitgebreide grafische gebruikersinterface dit proces wat vergemakkelijken.

Het uiteindelijke resultaat is een nieuwe, uitgebreide ontologie die informatie van beide originele ontologieën bevat. Deze is lokaal zodat er verder op gewerkt kan worden zoals op elke andere ontologie. Er kan bijvoorbeeld gebruik gemaakt worden van Sesame om een query uit te voeren op RDF-data.

1.3 Probleemstelling

De context van dit onderzoek is het vinden van een manier om het Semantisch Web zodanig te gebruiken zodat de eindgebruiker er voordeel uit haalt. Het is dus de bedoeling rijkere informatie aan te bieden aan de eindgebruiker, zonder dat er al te veel aan reeds bestaande applicaties moet veranderen.

Het Semantisch Web heeft als eigenschap dat iedereen zijn eigen datamodel kan aanmaken op maat van de applicatie. Dit voordeel kan ook als nadeel bekeken worden: het wordt immers erg onoverzichtelijk op het Semantisch Web en er is geen interoperabiliteit tussen de verschillende modellen. De relaties tussen databronnen gaan verloren. Deze relaties kunnen bestaan tussen bronnen die over dezelfde of over verschillende domeinen gaan. Op dit vlak is het mogelijke de huidige informatie te verrijken: relaties zijn ook informatie.

Een belangrijk probleem hierbij, en de kern van dit onderzoek, is dat er rekening gehouden moet worden met het feit dat al deze data gedistribueerd opgeslagen is. Een gedistribueerde query-taal voor het Semantisch Web is nog niet ontwikkeld. Wat eerst een verrijking-verhaal was, krijgt nu een extra dimensie: gedistribueerd werken is aan de orde. Dit laat op zijn beurt andere vragen de kop op steken, vooral op het vlak van efficiëntie. Een gedistribueerd systeem speelt zich af in een vrij complexe omgeving waardoor er met factoren zoals netwerkbelasting, wachttijden, verdelen van het werk, enz. ook rekening gehouden moet worden.

1.4 Motivatie voor Distributie

De metadata op het Semantisch Web speelt een grote rol in het oplossen van dit probleem. Applicaties kunnen zelf redeneren over onderdelen van een ontologie. Zo kunnen bijvoorbeeld overeenkomsten uitgedrukt worden en opgezocht worden, twee belangrijke zaken in dit onderzoek.

Het grootste voordeel van het gedistribueerd werken zal de uiteindelijke soepelheid zijn waarmee het systeem aangepast kan worden: er kunnen namelijk makkelijk nieuwe databronnen bijgevoegd worden. Het is de opzet van dit onderzoek een framework aan te bieden aan ontwikkelaars om op een makkelijke manier om te gaan met distributie en queries uit te voeren over gedistribueerde databronnen. Een nieuwe laag boven deze bestaande applicaties kan dan queries uitvoeren op deze data. Deze laag is een combinatie van:

- een web-applicatie: er wordt gedistribueerd gewerkt

- een database-applicatie: er wordt informatie opgevraagd uit een gegevensbestand
- het Semantische Web: metadata wordt gebruikt om relaties tussen repositories vast te leggen.

Er is al heel wat terug te vinden over het relateren van ontologieën. Wat steeds terugkomt is dat er vooral ontologieën geïntegreerd worden, of nieuwe ontologieën aangemaakt worden. Dit heeft als gevolg dat alles gecentraliseerd wordt en dat levert dan weer problemen op met betrekking tot updates, zeker wanneer er gewerkt wordt op databronnen die frequent aangepast worden. Het is de opzet in dit onderzoek om in de geest van het Semantisch Web te werken en alle data gedistribueerd te behouden. Het nadeel hiervan is dat het netwerk vaker belast wordt, maar mits een goede werkwijze wordt de hoeveelheid data die over het netwerk moet verstuurd worden, tot een minimum gereduceerd.

Bij alle theorieën die reeds vermeld zijn, bevindt het uiteindelijk resultaat zich op 1 bepaalde locatie. In deze oplossingen wordt er bottom-up gewerkt: een nieuwe ontologie wordt aangemaakt en lokaal bijgehouden. In dit document wordt er top-down gewerkt: de input komt van de gebruiker, wordt ontleed en dan gedistribueerd behandeld. De nadruk ligt niet enkel op het relateren van data uit verschillende bronnen, maar ook op het effectief uitvoeren van queries zonder alle data volledig te moeten kopiëren over een netwerk.

De meerwaarde is dat er rijkere informatie aangeboden wordt naar de eindgebruikers toe en dat er aan de ontwikkelaars, die van dit systeem gebruik maken om een applicatie te ontwikkelen, een tool wordt aangeboden om informatie van gedistribueerde databronnen op te halen. Zo kunnen bijvoorbeeld bij een zoekactie naar een bepaald onderwerp, eerst alle synoniemen van dit onderwerp opgezocht worden om hiervan ook de zoekresultaten te tonen. Dit systeem kan ook gebruikt worden om zoekresultaten te personaliseren. In de volgende hoofdstukken zullen nog voorbeelden uitgewerkt worden.

Er kan geconcludeerd worden dat er nood is aan een systeem dat toelaat relaties tussen verschillende databronnen te definiëren en het dan ook mogelijk maakt om van deze verrijkte data informatie op te vragen. Bij dit alles moet de gedistribueerde aard gerespecteerd blijven. Deze eis laat dan ook niet toe dat er een tijdelijke ontologie lokaal geproduceerd wordt die informatie uit alle bronnen samenvoegt.

1.5 Doelstellingen

1.5.1 Opbouw van het systeem

Bij de opbouw van DiSemDa moet er toch op enkele zaken gelet worden. Het is de bedoeling een generieke architectuur te construeren die in staat stelt om op gepaste wijze om te gaan met distributie. Een eis is bijvoorbeeld dat de gedistribueerde natuur van het Semantisch Web behouden moet worden. Het is niet de bedoeling gehele repositories over het netwerk te verzenden.

Bovendien is het ook belangrijk dat de menselijke interactie tot een minimum beperkt blijft. Er zal steeds een configuratie-luik aan te pas komen, dit is niet te vermijden. De configuratie moet dus zo weinig mogelijk inspanning vragen.

Rekening houdend met het feit dat het om een gedistribueerd systeem gaat, moeten er ook enkele performantie-vraagstukken beantwoord worden in dit onderzoek. De totale uitvoeringstijd hangt van meer af dan enkel de lokale performantie, ook bijvoorbeeld van netwerkbelasting, eigenschappen van de servers, enz.

Modulariteit is ook een belangrijk punt. Het eindresultaat moet toepasbaar zijn in zoveel mogelijk situaties. Daarom moet het modulair opgebouwd zijn zodat de niet-generische delen vervangbaar zijn.

1.5.2 Behandelen van queries

Queries zijn het startpunt van het hele proces. Deze moeten dan ook ontleed kunnen worden om gedistribueerd uitgevoerd te kunnen worden op het Semantisch Web. Hiervoor wordt zoveel mogelijk de eigenschappen van de queries uitgebuit. Eens verschillende subqueries opgebouwd zijn, moeten deze nog op verschillende servers uitgevoerd worden via het Internet.

Er moet ook aandacht besteed worden om na te gaan of de volgorde van uitvoering van subqueries belangrijk is voor de performantie.

Een heel belangrijke stap in dit onderzoek is het verrijken van queries. Maar hoe moet dit juist gebeuren? Er wordt getracht onafhankelijk te werken van specifieke API's en enkel gebruik te maken van een query-taal.

1.5.3 Overig

De voorstelling van een link tussen ontologieën moet voorgesteld kunnen worden. Het is niet de bedoeling in dit onderzoek om hierin even ver te gaan als MAFRA met de Semantic Bridge Ontology. De voorstelling van een link is één zaak, maar die link moet ook nog effectief kunnen uitgevoerd worden. Wanneer nieuwe resultaten uit deelresultaten aangemaakt worden, is ook bepalend voor de performantie.

Communicatie tussen gebruiker en systeem is belangrijk. De input die de gebruiker geeft moet zo simpel mogelijk gehouden worden, bij voorkeur iets dat al gekend is zoals een reeds bestaande query-taal. Nieuwe, samengevoegde resultaten moeten op een duidelijke manier naar de gebruiker worden teruggegeven. De vorm moet zo zijn dat er niet bij stilgestaan moet worden dat het van gedistribueerde bronnen afkomstig is.

1.6 Voorgestelde Oplossing

De huidige versie van DiSemDa is ontwikkeld om mee te experimenteren, om een indruk te krijgen van het belang van de configuratie van de distributie. Tijdens het ontwikkelingsproces was er een samenwerking met Mathieu Cardinael.

DiSemDa is een query-server. De gebruikers in de eerste rang zullen dus ontwikkelaars zijn. De ontwikkelaar kan meerdere repositories toevoegen aan het systeem en relaties tussen concepten vastleggen. Verschillende repositories worden met elkaar verbonden op basis van semantische gelijkheid van 2 concepten, dit is een mapping. Meer configuratie is er niet nodig, ook niet bij het uitvoeren van een query.

Wanneer de ontwikkelaar een SeRQL-query als input geeft, wordt deze eerst ontleed tot verschillende subqueries. De subqueries worden vervolgens uitgevoerd, maar er wordt rekening gehouden met de mappings. Eerst wordt er namelijk een subquery gestuurd naar de eerste repository binnen een mapping. Vervolgens is de tweede repository aan de beurt, maar deze query wordt verrijkt met informatie uit de resultaten van de eerste query. Zo wordt er enkel relevante informatie opgevraagd uit de tweede repository. De deelresultaten worden samengevoegd zodat er 1 resultaat weergegeven wordt. Aan mappings worden prioriteiten meegegeven zodat de ontwikkelaar volledige controle heeft over de volgorde van uitvoering van de subqueries.

De vorm van het resultaat dat teruggegeven wordt naar de ontwikkelaar toe is zodanig dat het niet merkbaar is dat de resultaten van verschillende repositories afkomstig zijn. Dit bevordert het gebruiksgemak.

Er is gekozen voor een "client-server"-architectuur omwille van performantieredenen: hele repositories kopiëren is immers geen schaalbare oplossing. Voor het server-deel wordt Sesame Server gebruikt. Deze module geeft de mogelijkheid over het Internet een SeRQL-query uit te voeren. Het client-gedeelte bestaat uit het voorbereiden van de subqueries en het verwerken van de resultaten van de subqueries.

Dynamiek is een belangrijke eigenschap van DiSemDa. Er kunnen ten allen tijde repositories en mappings bijgevoegd of verwijderd worden. DiSemDa heeft een modulaire opbouw zodat er in de toekomst nog veel uitbreidingsmogelijkheden zijn.

2

Distributed Semantic Data

In de vorige sectie is uitgelegd waarom er nood is aan een systeem dat gedistribueerde queries op het Semantisch Web kan uitvoeren. Na vele denkoefeningen is "DiSemDa"[3] tot stand gekomen. Dit hoofdstuk bespreekt de gehele structuur van het DiSemDa-project.

De functionaliteit die DiSemDa aanbiedt zal besproken worden. Specifieke eisen hebben geleid tot een design en deze overwegingen in het design worden eveneens uitvoerig besproken. Vervolgens wordt de werking van DiSemDa in detail uitgelegd.

[3] **Di**stributed **Sem**antic **Da**ta

2.1 DiSemDa Functionaliteit

Bij het ontwikkelen van DiSemDa is er geprobeerd om de voordelen van data-integratie te combineren met de voordelen van gedistribueerd werken. Voordelen van data-integratie zijn:

- alles is via 1 centraal portaal toegankelijk
- de respons-tijd is niet afhankelijk van de netwerkbelasting of snelheid van andere servers, maar enkel van de eigen uitvoeringstijd
- relaties tussen data kunnen ten volle uitgebuit worden

Aan gedistribueerd werken zijn ook voordelen verbonden:

- elke databron bestaat onafhankelijk van de andere en is dus ook niet onderhevig aan het falen van andere bronnen
- de belasting van servers wordt verdeeld, met als gevolg dat er communicatie tussen servers nodig is en dat het netwerk ook belast wordt
- een gedistribueerd systeem is dynamischer omdat het niet veel moeite kost data van een reeds bestaande databron op te nemen in het systeem
- elke databron is op maat gemaakt van de applicatie

DiSemDa is een portaal naar verschillende gedistribueerde repositories. De gebruiker zal hier echter niets van merken: het lijkt alsof men informatie van 1 centrale repository queried. DiSemDa neemt 1 SeRQL-query aan van de gebruiker, splitst deze query op en stuurt de subqeries naar de verschillende repositories. Achteraf worden de resultaten samengevoegd en als 1 resultaat teruggegeven naar de gebruiker.

DiSemDa draait rond het verbinden van verschillende repositories, die data én metadata bevatten, met elkaar. Er is mogelijkheid tot het configureren van het systeem zodat het aangepast kan worden om het meest efficiënt te werken in welke situatie dan ook.

Sommige zaken zijn eigen aan een gedistribueerd systeem en zijn dus niet te verzoenen met voordelen van een gecentraliseerde aanpak. Zo zal de uitvoeringstijd steeds afhangen van de belasting van het netwerk en de verschillende servers. Maar mits een goed design kan dit laatste probleem tot een minimum herleid worden.

De voordelen wegen op tegen de nadelen. DiSemDa is een zeer dynamisch systeem dat toe laat gedistribueerd te werken zonder dat de gebruiker dit merkt. Alle voordelen van een gedistribueerd systeem worden behouden. Zo blijven alle repositories onafhankelijk van elkaar en wordt de systeembelasting verdeeld. Dit, samen met het kunnen gebruiken van relaties tussen data, zijn de grootste voordelen van DiSemDa. Van in het begin was het duidelijk dat volledige repositories kopiëren over een netwerk en hiervan lokale kopieën bijhouden, geen mogelijkheid was. Deze kopieën moeten dan steeds consistent gehouden worden met de originele repositories. Dit probleem kan vermeden worden door een wat meer doordachte aanpak.

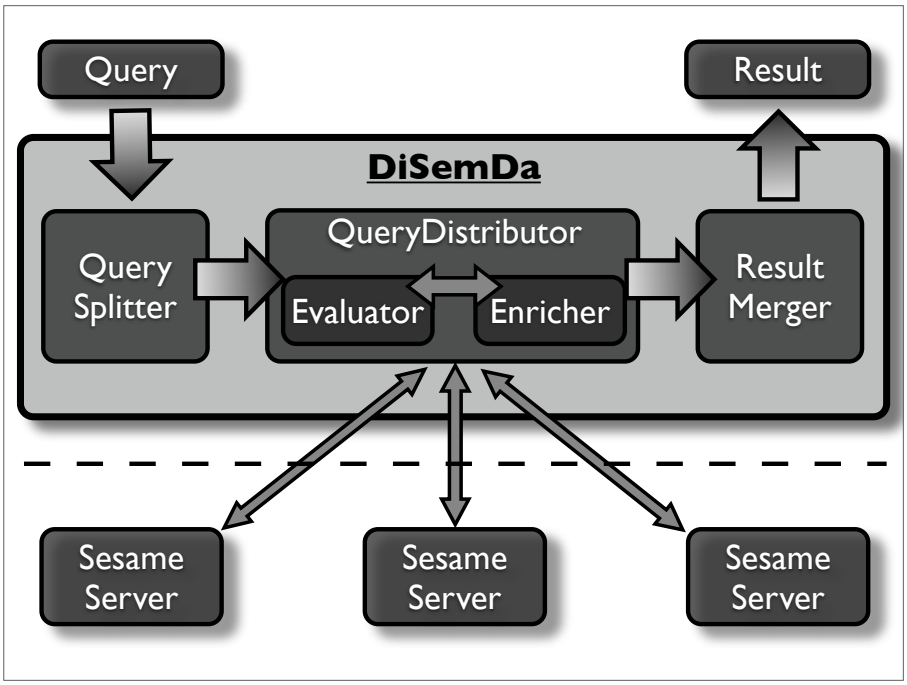
DiSemDa gaat een stap verder dan MAFRA. Waar MAFRA enkel aandacht schenkt aan het specificeren van relaties en het aanmaken van een nieuwe ontologie, zorgt DiSemDa ervoor dat er ook effectief queries kunnen uitgevoerd worden. Dit zonder het aanmaken van een nieuwe ontologie. In een omgeving waar inhoud van repositories snel verandert, is dit een belangrijk voordeel. Het is wel zo dat MAFRA toelaat relaties veel specifieker te definiëren, maar in de sectie "Toekomstig Werk", wordt er uitgelegd hoe er op dit vlak verbeteringen kunnen aangebracht worden zodat DiSemDa aangepast kan worden naargelang de situatie.

2.2 DiSemDa Design

2.2.1 Globaal Overzicht

DiSemDa is een systeem dat het best te vergelijken valt met een query-server. Verder in deze subsectie zal de samenhang tussen de verschillende onderdelen uitgelegd worden. Elk onderdeel wordt in de volgende subsectie in detail besproken.

Figuur 5 is een weergave van DiSemDa. De stippellijn geeft het verschil aan tussen wat lokaal gebeurt en wat via het internet gebeurt. Het is ook duidelijk te zien dat DiSemDa gebruik maakt van Sesame[9] om queries uit te voeren op een bepaalde repository. Sesame bevat immers een web-interface die de mogelijkheid geeft via het Web een query uit te voeren.



Figuur 5: Globaal design DiSemDa

De input voor DiSemDa is een SeRQL-query. Aan de hand van de eigenschappen deze query, en de informatie uit de configuratie, wordt deze query opgesplitst in verschillende subqueries door de QuerySplitter

De QueryDistributor zorgt er vervolgens voor dat een subquery naar de juiste repository gestuurd wordt. Hier wordt er rekening gehouden met Mappings. Een Mapping is een object dat de relatie tussen 2 repositories voorstelt. De mappingvariabele is de variabele in de originele query die de mapping belichaamt. Deze variabele komt minstens in 2 verschillende subqueries voor. De verschillende waarden van de mappingvariabele van de resultaten van de ene repository, worden door de QueryEnricher verwerkt in de subquery die bestemd is voor de andere repository in dezelfde mapping. Dit zorgt ervoor dat deze laatste query selectiever is, bijgevolg zal het resultaat ook kleiner zijn. Dit heeft ook een minimale winst op het gebied van netwerkbelasting tot gevolg, maar de winst is bij het samenvoegen, de volgende stap, is beter waarneembaar. De QueryEvaluator zorgt dat een query met behulp van Sesame[9] uitgevoerd wordt.

Ten slotte zorgt de ResultMerger ervoor dat de resultaten uit de verschillende repositories samengevoegd worden zodat het lijkt alsof de tupels die teruggegeven worden aan de gebruiker afkomstig zijn van een lokale repository. Dit gedeelte is intensief qua rekenkracht en daarom is het dus belangrijk om de input voor deze laatste stap zo klein mogelijk te maken.

DiSemDa is een dynamisch systeem. Op elk moment kunnen er nieuwe repositories of nieuwe mappings op reeds geconfigureerde repositories, toegevoegd worden. DiSemDa in zijn huidige vorm, voert mappings uit op gelijkheid, maar het systeem is opgebouwd zodat ook dit aanpasbaar is. Meer hierover in "DiSemDa Doeleinden" (3.2).

2.2.2 Design Overwegingen

Tijdens het opmaken van het design zijn er heel wat dingen afgewogen. Zo werd het meeste aandacht besteed aan schaalbaarheid, efficiëntie en dynamiek. De efficiëntie van een gedistribueerd systeem wordt bepaald door verschillende factoren. De belangrijkste zijn het aantal queries die verstuurd worden, de hoeveelheid data dat verstuurd wordt over het netwerk en de lokale kost om de join uit te voeren.

De grootste onderzoeksvraag was wanneer het best de nieuwe, samengevoegde tupels binnen een mapping aangemaakt zouden worden. Er zijn verschillende mogelijkheden.

Lokale Kopieën

De eerste mogelijkheid is om dit te doen voordat er queries zouden uitgevoerd worden. Deze aanpak bestaat erin om eerst beide repositories lokaal te kopiëren. Vervolgens worden beide repositories gejoined op basis van de waarde van de mappingvariabele. Er wordt dus een nieuwe ontologie gevormd. Dit is heel analoog met de werking van MAFRA[5].

Er is een mogelijkheid om aan performantie te winnen. Zo zouden de kopieën lokaal kunnen bijgehouden worden om te gebruiken voor de volgende queries. Die richting mondde echter uit in een consistentie-probleem: hoe worden de lokale kopieën consistent gehouden met de originele repositories?

Deze aanpak is niet schaalbaar en zeer inefficiënt, zeker wanneer er sprake is van meerdere repositories met een grote inhoud. Er wordt onnodig veel data gekopieerd en dus zou het netwerk ook meer dan nodig belast worden. Deze "overload" zijn de tupels die geen match vinden in de andere repository binnen een mapping.

Andere mogelijke theorieën houden geen lokale kopieën bij en hebben dus ook geen last van het consistentie-probleem. Bovendien wordt een join uitgevoerd tussen alle data van beide repositories, wat ook helemaal niet altijd nodig is: wanneer er een query opgesteld wordt dat alle concepten opvraagt met een bepaalde waarde voor een eigenschap, is het voldoende enkel deze concepten met de juiste waarde voor deze eigenschap te betrekken in de join-operatie.

Deze werkwijze wordt verduidelijkt aan de hand van een voorbeeld. Voorbeeld 1 geeft de inhoud van 2 repositories: een restaurant-repository, met de naam van een restaurant en de postcode, en een cinema-repository, met de naam van een bioscoop en de postcode. Deze informatie wordt samengevoegd zodat restaurants en bioscopen met dezelfde zipcode aan elkaar gekoppeld worden. Zo wordt het duidelijk welke restaurants in de nabijheid van bioscopen liggen.

Alle data uit beide repositories worden lokaal gekopieerd. Vervolgens moet elk tupel van de restaurant-repository verbonden worden met elk tupel van de cinema-repository met dezelfde zipcode. De overload is geïllustreerd door restE met zipcode 8020 en cineC met zipcode 1640. In dit geval zijn het slechts 2 tupels, maar in realiteit komt deze situatie vaker voor.

Voorbeeld 1

restaurant repository	
name	zipcode
restA	1500
restB	1500
restC	1500
restD	8000
restE	8020

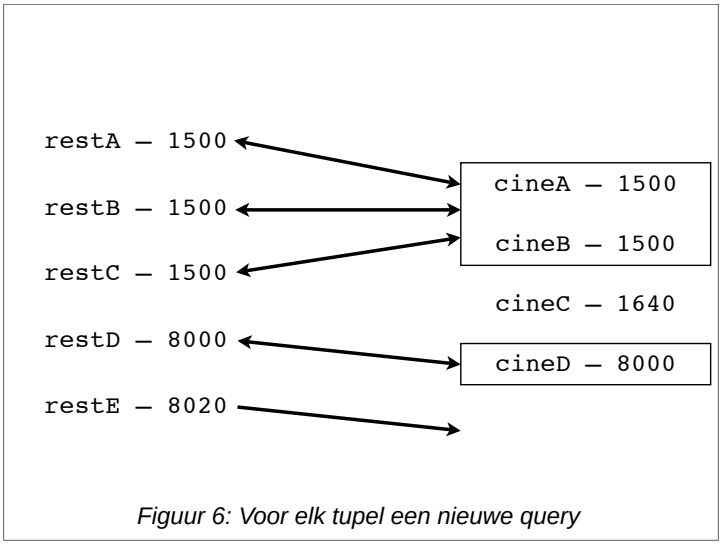
cinema repository	
name	zipcode
cineA	1500
cineB	1500
cineC	1640
cineD	8000

Voor Elk Tupel Een Nieuwe Query

Een tweede aanpak probeert het netwerk te sparen. Er worden helemaal geen lokale repositories bijgehouden. De eerste repository binnen een mapping wordt gequeried. Voor elk tupel uit dit resultaat, wordt een nieuwe hulp-query opgesteld, maar deze hulp-query is wel verrijkt met de waarde van de mappingvariabele van dit tupel. Het nieuw resultaat uit de tweede repository moet gejoined worden met het eerste tupel. Dit wordt voor elk tupel gedaan uit het eerste resultaat.

Een voordeel van deze aanpak is dat resultaten onmiddellijk kunnen samengevoegd worden, omdat men, door de verrijking, weet welke tupels samengevoegd moeten worden op basis van de mappingvariabele. Nog een voordeel is dat er geen overload is in de resultaten uit de tweede repository. Een nadeel is dat er veel queries over het netwerk gestuurd worden. Bovendien zullen ook veel queries hetzelfde resultaat teruggeven, dit betekent een onnodige overlast voor het netwerk en de tweede server.

Deze werkwijze wordt geïllustreerd in figuur 6. Deze figuur is gebaseerd op voorbeeld 1. Elke pijl stelt een hulp-query voor. Elke hulp-query is verrijkt met een zipcode. Daarom is het ook normaal dat de eerste 3 hulp-queries hetzelfde resultaat teruggeven: de eerste 3 triples uit de eerste repository hebben immers dezelfde zipcode. De server van de bioscoop repository moet 3 keer dezelfde hulp-query uitvoeren en het netwerk wordt ook 3x met hetzelfde resultaat belast wordt. De laatste pijl stelt een hulp-query voor die geen resultaat teruggeeft, maar toch moet er een hulp-query worden gestuurd. Men kan dus concluderen dat 3 van de 5 queries die getoond worden in figuur 6 overbodig zijn.



Figuur 6: Voor elk tupel een nieuwe query

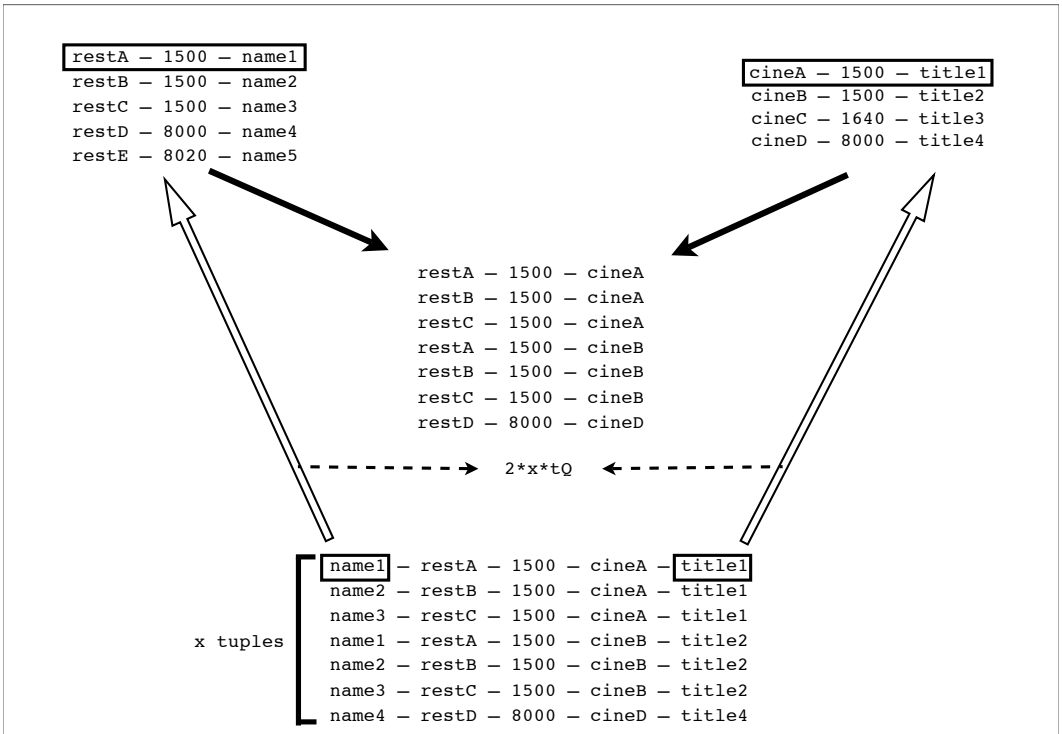
Minimum Overload

Er is nog een andere werkwijze, meer gefocust op het minimaliseren van overload. Het verzamelen van informatie wordt opgesplitst in 2 delen. Een eerste stap focust zich op het aanmaken van nieuwe tupels en is analoog met de werkwijze waarin lokale kopieën gebruikt worden. Van elke node wordt de identiteit en de waarde van de mappingvariabele opgevraagd. Dit gebeurt voor beide repositories. De tupels van deze resultaten worden gejoined tot nieuwe tupels. Dan komt de tweede stap. Deze nieuwe, samengevoegde tupels zijn immers nog niet volledig, enkel essentiële informatie is in de eerste stap opgevraagd. De andere informatie die nodig is, moet nog opgehaald worden. Voor elk samengevoegd tupel moeten dus opnieuw 2 hulp-queries gestuurd worden: 1 naar elke repository.

Deze werkwijze minimaliseert de overload. Er is nog steeds overload want in de eerste stap worden alle tupels verkregen, maar aangezien enkel de essentiële elementen opgehaald zijn, is de overload per tupel minimaal. Er worden er in de eerste stap nog altijd teveel tupels opgevraagd uit beide repositories. Het grootste nadeel is dat er in de tweede stap erg veel hulp-queries gestuurd worden, zeker wanneer het een niet-selectieve gedistribueerde query betreft. Bovendien heeft de tweede stap hetzelfde nadeel als de werkwijze waarin voor elk tupel een nieuwe hulp-query gestuurd wordt: er is veel kans dat er meerdere keren dezelfde hulp-query gestuurd met als gevolg onnodige belasting van de servers en het netwerk.

Een voorbeeld verduidelijkt (figuur *7*). Er wordt nog steeds gewerkt met de repositories uit voorbeeld 1. In de eerste stap worden de essentiële zaken opgevraagd uit beide repositories, in dit geval bv. het tupel "restA—1500" voor de restaurant-repository en voor de bioscoop bv. het tupel "CineB—1500". Deze tupels worden samengevoegd tot x nieuwe tupels. Dan is het in de tweede stap nodig $2*x$ queries te sturen om deze tupels uit te breiden (niet ingekleurde pijlen), in dit geval met de naam van een restaurant en de titel van een bioscoop. De uitvoeringstijd van de tweede stap is dus $2*x*tQ$ met tQ de uitvoeringstijd van een query.

In het voorbeeld is duidelijk te zien dat "name1", "name2" en "name3" elk 2 keer worden opgevraagd en "title1" en "title2" elk 3 keer worden opgevraagd. De servers worden dus verschillende keren belast met dezelfde hulp-query, alsook het netwerk dat dubbele resultaten moet vervoeren. De lokale overload wordt bij deze aanpak wel geminimaliseerd, maar de netwerk- en serverbelasting duidelijk niet.



Figuur 7: Systeem met minimale overload

Als de uitvoeringstijd mee in de vergelijking wordt genomen, is het ook duidelijk dat deze werkwijze niet het snelst een resultaat zal opleveren. Deze methode zal zeer snel een heel grote uitvoeringstijd hebben, zeker bij grote repositories. Figuur 8 is een meting van een gedistribueerde query waar alle repositories zich lokaal bevonden. Er moet dus geen rekening gehouden worden met het netwerk en kloksnelheid van servers in dit voorbeeld. Het is daar duidelijk te zien dat de hulp-queries een zeer groot aandeel hebben in de uitvoeringstijd. Als er dan *2*x* keer meer hulp-queries moet uitvoeren, wordt de uitvoeringstijd natuurlijk heel snel veel groter.

```
time split: 142 ms
mapping priority : 4
=> SELECT zipcode FROM {card} vcard:Pcode {zipcode} USING
time query: 173 ms
=> SELECT DISTINCT cityname, cinename, zipcode FROM {cine
time query: 37 ms
mapping priority : 1
=> SELECT DISTINCT restoname, zipcode FROM {resto} resto-
time query: 1028 ms
time execute : 1244 ms
time merge 2 ms
total time : 1389 ms
```

Figuur 8: Aandeel van queries in de uitvoeringstijd

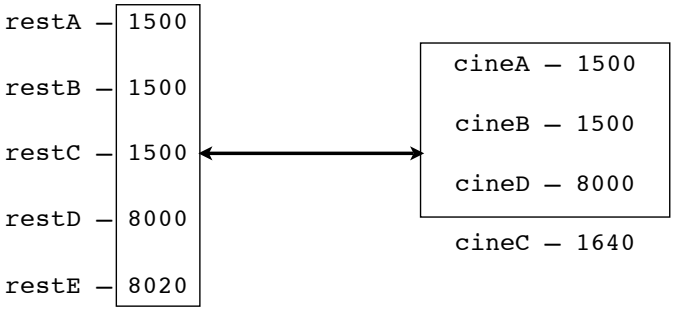
Een Compromis

Er moet een compromis gezocht worden tussen overload enerzijds en netwerk- en serverbelasting anderzijds. Het compromis bestaat erin een subquery uit te voeren op de eerste repository binnen een mapping. Er wordt onmiddellijk alle nodige informatie opgehaald. Alle verschillende waarden van de mappingvariabele wordt uit het resultaat gehaald. Er wordt 1 query voor de tweede repository verrijkt met al deze waarden. Het tweede resultaat bevat dus geen overload.

De juiste tupels moeten wel nog samengevoegd worden, aangezien niet geweten is welke een match vormen. Dit is het grootste nadeel aan deze werkwijze. Daarom worden eerst alle tupels van het eerste resultaat overlopen. Voor elk tupel worden alle tupels van het tweede resultaat overlopen om te zien of er al dan niet een match is op basis van de mappingvariabele. Indien er wel een match is, worden beide tupels samengevoegd en toegevoegd aan het eindresultaat. Deze laatste stap is dezelfde stap als de aanpak die werd uitgelegd in 2.2.2.1. Dit is dus een minder efficiënte stap, maar de dataset is nu veel kleiner.

Er worden slechts 2 queries gestuurd bij deze aanpak, dit is onafhankelijk van de grootte van de repositories of het aantal resultaten. Er is natuurlijk ook een keerzijde. Zo is er nog steeds overload in het resultaat van de eerste repository. Maar er zal altijd wel ergens overbodige data zitten. Het is immers niet geweten voor welke waarden van de mappingvariabele de mapping tussen beide repositories kan gebeuren. Het voordeel is wel dat men door deze informatie, overbodige elementen uit het tweede resultaat kan weerhouden.

Rekening houdend met het feit dat we gedistribueerd werken, is deze aanpak het beste compromis dat gemaakt kan worden: een noodzakelijk minimum aan overbodige data wordt over het netwerk verstuurd, het netwerk wordt niet belast met dubbele resultaten, en er wordt slechts één query naar elke repository gestuurd.



Figuur 9: Voor alle resultaten samen slechts 1 nieuwe query

Figuur 9 is een voorbeeld van deze werkwijze, gebaseerd op voorbeeld 1. Er is slechts 1 pijl getekend, en dit houdt in dat er slechts 1 query gestuurd wordt naar de tweede repository. Deze query is wel verrijkt met alle waarden van de zipcode van het eerste resultaat. Het is wel zo dat er in de eerste query nog een overbodig resultaat zit, namelijk het tupel met zipcode 8020.

2.3 DiSemDa Ontrafeld

2.3.1 Configuratie van het Systeem

De configuratie van DiSemDa is minimaal. Er zijn 2 stappen die nodig zijn om DiSemDa te laten functioneren. Ten eerste moet er bepaald worden waar de repositories, waar queries naar gestuurd kunnen worden, zich bevinden. Dit is de URL langs waar Sesame web-toegang verleent. De naam van de repository moet ook meegegeven worden opdat er queries kunnen uitgevoerd worden. In dezelfde stap wordt ook de preprefix[4] voor een repository bepaald.

[4] zie 2.3.2

Vervolgens is het nodig mappings te definiëren. Een mapping vormt de verbinding tussen 2 repositories. Voor elke repository in een mapping is het nodig een combinatie van een namespace en RDF-predikaat mee te geven. Deze 2 predikaten worden later met elkaar moeten worden. Vervolgens moet aan een mapping ook een prioriteit meegegeven worden. Repositories uit mappings met een hogere prioriteit worden gequeried vòòr repositories uit mappings met een lagere prioriteit. Het is niet nodig dat deze prioriteiten opeenvolgend zijn of bij een bepaald getal moeten beginnen. Ook binnen een mapping ligt de volgorde van de uitvoering vast: de eerste repository binnen de mapping wordt eerst uitgevoerd, daarna de tweede. Door deze aanpak kan de gebruiker de volledige volgorde van uitvoering vastleggen.

Een voorbeeld van een mapping kan voorbeeld 2 zijn. Er wordt een brug gevormd tussen een bioscoop-repository en een restaurant-repository. Dit gebeurd op basis van de postcode waarin een restaurant of bioscoop zich bevindt. De RDF-predikaten zijn in beide gevallen "Zip", maar deze kunnen evengoed verschillen. Dit hangt af van de ontologie die gebruikt is. Met deze configuratie worden restaurants en bioscopen met elkaar verbonden op basis van hun ligging.

Voorbeeld 2

```
namespace 1:        http://www.vub.ac.be/cinema#
predikaat 1:        Zip
namespace 2:        http://chefmoz.org/rdf/elements/1.0#
predikaat 2:        Zip
prioriteit :        1
```

2.3.2 Opsplitsen van Queries

Werkwijze

DiSemDa krijgt als input een SeRQL-query, deze is niet verschillend van een query op een centrale repository. Er is slechts 1 extra spelregel die gevolgd moet worden: elke prefix die aan een namespace gekoppeld is, kan slechts tot 1 repository behoren. Indien meerdere repositories gebruik maken van eenzelfde namespace, is het nodig verschillende prefixen te definiëren in de "using namespace"-clause. De reden hiervoor is dat er "preprefixes" ingevoerd zijn om te kunnen bepalen bij welke repository een subquery hoort. Dit wordt verder in deze sectie uitgelegd.

Aan de hand van de vorm van de query, wordt deze in verschillende subqueries opgesplitst. Het splitsen start bij het laatste deel van de query: de "using-namespace"-clause. Deze clause bevat koppels van prefixes en namespaces. Aangezien een repository van meerdere namespaces gebruik kan maken, is het nog niet geweten bij welke repository een namespace hoort. Daarom zijn "preprefixes" ingevoerd. Dit is het eerste deel voor de "-" van de prefix. Elke repository heeft zijn eigen, unieke preprefix. Op deze manier kunnen namespaces, en later ook subqueries, aan repositories gekoppeld worden. Het eindpunt van deze stap is het opbouwen van een tabel die preprefixes koppelt aan prefix-definities.

De tweede stap in het opsplitsen is de "from"-clause onder handen nemen. Er wordt bepaald bij welke subquery een triple hoort door de preprefix die in dat triple gebruikt wordt. Ook in deze stap worden koppels opgebouwd tussen de preprefix en het triple. Maar deze triples bevatten nog meer informatie. Hier staan namelijk ook de verschillende variabelen in combinatie met de preprefix waarbij ze horen. Deze informatie is nuttig om in de "select"-clause en "where"-clause te bepalen bij welke repository een variabele hoort. Daarom worden er dus ook koppels gemaakt tussen, enerzijds de variabele en anderzijds de repository waar het bijhoort.

Op dit moment kan ook een mappingvariabele berekend worden. Deze komt namelijk voor in 2 verschillende subqueries en bijgevolg dus in combinatie met 2 verschillende preprefixes. Dit voelt ook heel natuurlijk aan, want wanneer er in SeRQL verschillende malen dezelfde variabele voorkomt in een query, neemt deze variabele steeds dezelfde waarde aan binnen hetzelfde tupel. Deze variabele is later belangrijk wanneer queries verrijkt worden (2.3.3) en resultaten samengevoegd worden(2.3.4).

De "select"-clause en "where"-clause kunnen op dezelfde manier behandeld worden. Eerst wordt een clause onderverdeeld in verschillende delen en dan wordt er uit elk deel de variabele geëxtraheerd. Vervolgens wordt er opgezocht bij welke repository de variabele hoort in de koppels die opgebouwd zijn uit de "from"-clause. Voor elke clause worden er apart koppels aangemaakt tussen de preprefix en het onderdeel van de clause die de variabele in kwestie bevat.

Een preprefix is een identificatie voor een repository. Alle koppels die in voorgaande stappen opgebouwd zijn bevatten deze identificatie. Uit deze koppels kan per preprefix een volledige subquery opgebouwd worden.

Elk onderdeel van een query wordt slechts 1 maal overlopen. De informatie wordt lokaal in tabellen bijgehouden. Dit levert een performantiewinst op, vooral bij de "select"- en "where"-clause. Hier is immers informatie nodig uit de "from"-clause. Deze tabellen worden reeds opgebouwd wanneer de "from"-clause overlopen wordt.

Voorbeeld

Een query staat neergeschreven in voorbeeld 3 die een bioscoop-repository met een restaurant-repository combineert. De restaurant-repository bevat restaurant-gegevens zoals een naam, het adres en het soort eten dat geserveerd wordt. De bioscoop-repository is gevuld met bioscoop-informatie zoals het adres en de naam van de bioscopen. De query in dit voorbeeld gaat op zoek naar alle bioscopen in de stad Brugge waar een Italiaans restaurant in dezelfde postcode ligt.

Voorbeeld 3

```
SELECT  zipcode, cinename, cityname, restoname
FROM  {cine} cine:City {cityname},
      {cine} cine:Name {cinename},
      {cine} cine:Zip {zipcode},
      {resto} resto-d:Title {restoname},
      {resto} resto-r:Cuisine {cuisinetype},
      {resto} resto-r:Zip {zipcode}
WHERE cityname LIKE “Brugge”,
      cuisinetype LIKE “Italian”
USING NAMESPACE
      cine=<http://www.vub.ac.be/cinema#>,
      resto-r=<http://chefmoz.org/rdf/elements/1.0#>,
      resto-d=<http://purl.org/dc/elements/1.0/>
```

Er zijn 2 namespaces waarvan de restaurant-repository gebruik maakt, nl. `http://chefmoz.org/rdf/elements/1.0#` en `http://purl.org/dc/elements/1.0/`. De ”prefixen” die hierbij horen zijn `resto-r` en `resto-d`. Aangezien deze prefixen dezelfde preprefix bevatten, nl. “`resto`”, kan het systeem bepalen dat beide namespaces bij dezelfde restaurant-repository horen. Dit is geïllustreerd in voorbeeld 4.

De preprefixen van de onderdelen die tot de subquery zullen behoren die naar de restaurant-repository zullen gestuurd worden, zijn onderlijnd, de preprefixen van de onderdelen die naar de bioscoop-repository zullen gestuurd worden, zijn vet en schuin gedrukt.

Voorbeeld 4

```
USING NAMESPACE
      cine=<http://www.vub.ac.be/cine#>,
      resto-r=<http://chefmoz.org/rdf/elements/1.0#>,
      resto-d=<http://purl.org/dc/elements/1.0/>
```

Vervolgens wordt de "from"-clause ontleed. De eerste 3 triples behoren tot de subquery voor de bioscoop-repository en de volgende 3 tot de subquery voor de restaurant-repository omdat ze respectievelijk de preprefix "cine" en "resto" hebben (voorbeeld 5). Er moet ook nog bepaald worden bij welke preprefix een variabele hoort. In dit voorbeeld is dit zichtbaar door dat de variabelen cine, cityname, cinename en zipcode voorkomen in combinatie met "cine", terwijl de variabelen resto, restoname, cuisinetype en zipcode voorkomen in combinatie met "resto".

Voorbeeld 5

```
FROM  {cine} cine:City {cityname},
      {cine} cine:Name {cinename},
      {cine} cine:Zip {zipcode},
      {resto} resto-d:Title {restoname},
      {resto} resto-r:Cuisine {cuisinetype},
      {resto} resto-r:Zip {zipcode}
```

Daarna moet de "select"-clause opgesplitst worden. Voorbeeld 6 toont deze opdeling. Het is een kwestie van de naam van de variabele terug te vinden in een triple uit de "from"-clause. In de "where"-clause kan dezelfde techniek toegepast worden (zie voorbeeld 7).

Voorbeeld 6

```
SELECT  zipcode, cinename, cityname, restoname
```

Voorbeeld 7

```
WHERE cityname LIKE "Brugge",
      cuisinetype LIKE "Italian"
```

Volgens deze procedure worden er uiteindelijk subqueries bekomen, in dit voorbeeld 2 subqueries, die volledig onafhankelijk van elkaar kunnen uitgevoerd worden. Deze queries worden getoond in voorbeeld 8 en 9. Voorbeeld 8 is een subquery bestemd voor de bioscoop-repository, terwijl voorbeeld 9 bestemd is voor de restaurant-repository. Dit resultaat wordt verder gebruikt in het systeem.

Voorbeeld 8

```
SELECT cityname, cinename, zipcode
FROM {cine} cine:City {cityname},
      {cine} cine:Name {cinename},
      {cine} cine:Zip {zipcode}
WHERE cityname LIKE "Brugge"
USING NAMESPACE
      cine = <http://www.vub.ac.be/cinema#>
```

Voorbeeld 9

```
SELECT restoname, zipcode
FROM {resto} resto-d:Title {restoname},
      {resto} resto-r:Cuisine {cuisinetype},
      {resto} resto-r:Zip {zipcode}
WHERE cuisinetype LIKE "Italian"
USING NAMESPACE
      resto-r=<http://chefmoz.org/rdf/elements/1.0#>,
      resto-d=<http://purl.org/dc/elements/1.0/>
```

2.3.3 Evaluatie van Subqueries

Werkwijze

Het is nodig de subqueries in een heel specifieke volgorde uit te voeren. Ten eerste speelt de prioriteit van de mappings een rol. Eerst worden de mappings opgezocht die kunnen gebruikt worden voor de query in kwestie. De voorwaarde hiervoor is dat er een mappingvariabele moet zijn die in beide subqueries van de mapping voorkomt in een triple in de "from"-clause. De naam van deze variabele wordt bijgehouden in de mapping.

De mappings worden vervolgens in de juiste volgorde overlopen: van hoge naar lage prioriteit. De keuze voor deze volgorde is gevallen omdat het zo makkelijker is om een nieuwe mapping te laten plaatsvinden vòòr de reeds bestaande mappings. Hiervoor moet men dus enkel een nieuwe mapping met hogere prioriteit toevoegen. Dit is waarschijnlijk de vorm die het meest zal voorkomen. Dit is echter geen restrictie, men kan bij de configuratie genoeg ruimte laten om toe te laten mappings toe te voegen met een kleinere prioriteit dan de bestaande mappings.

Voor elke mapping wordt eerst de eerste query in de mapping uitgevoerd. Dit gebeurt aan de hand van Sesame[9]. De informatie waar een server zich bevindt en over welke repository het gaat, wordt uit de DiSemDa configuratie gehaald (zie 2.3.1). De waarden van de mappingvariabele van het resultaat van deze query worden opgeslagen in de mapping. Vervolgens wordt de tweede query uit de mapping uitgevoerd. Deze query wordt verrijkt met de waarden van de mappingvariabele uit het vorige resultaat.

De verrijking zit vervat in de "where"-clause: deze wordt uitgebreid met gelijkheidsoperatoren en de waarden van de mappingvariabele. Het voordeel hiervan is dat DiSemDa onafhankelijk is van de Sesame[9] API en daardoor dus ook aangepast kan worden zodat het werkt met een ander systeem. De enige requirement voor dergelijk systeem is dat een SeRQL-query uitgevoerd kan worden via het Internet.

Wanneer éénzelfde mappingvariabele voorkomt in verschillende mappings, worden de waarden van de mappingvariabele van de vorige mapping gebruikt om de volgende query te verrijken. Dit heeft ook als voordeel dat de verzameling van resultaten vanaf de tweede mapping nog kleiner zal zijn, omdat de query voor deze repository onmiddellijk verrijkt is met informatie uit vorige mappings. Dit beïnvloed de uitvoeringstijd positief. Op deze manier wordt ook vermeden dat een repository meerdere keren gequeried wordt: de reeds gevonden resultaten worden gebruikt. Het heeft geen zin van nog andere waarden voor de mappingvariabele op te vragen, want deze worden toch niet opgenomen in de join. Een voorbeeld hiervan wordt in het volgend hoofdstuk uitgewerkt (3.3.2).

Wanneer alle repositories, die in een mapping betrokken zijn, gequeried zijn, zijn de overige repositories aan de beurt. Dit gebeurt op dezelfde manier als voorheen uitgelegd, enkel moet er geen rekening gehouden worden met mappings. Het verrijken van een query is in dit geval dus ook niet aan de orde.

Op het moment dat alle subqueries uitgevoerd zijn, zitten alle resultaten in een tabel. Deze resultaten kunnen aangesproken worden via de preprefix die bij een repository hoort. Deze tabel is het eindresultaat van deze fase en wordt verder doorgegeven in het systeem om de resultaten samen te voegen.

Voorbeeld

Indien de configuratie uit voorbeeld 1 nog steeds van kracht is, wordt eerst de bioscoop-repository gequeried. Aan deze query wordt niets verandert. Vervolgens wordt de restaurant-repository gequeried, maar deze wordt wel verrijkt met alle verschillende waarden van de mappingvariabele uit de eerste repository. De zipcodes 8200, 8000 en 1500 in voorbeeld 10 zijn deze waarden uit de bioscoop-repository.

Voorbeeld 10

```
SELECT DISTINCT restoname, zipcode
FROM {resto} resto-d:Title {restoname},
      {resto} resto-r:Cuisine {cuisinetype},
      {resto} resto-r:Zip {zipcode}
WHERE cuisinetype LIKE “Italian”
      AND (zipcode LIKE "8200" IGNORE CASE
      OR zipcode LIKE "8000" IGNORE CASE
      OR zipcode LIKE "1500" IGNORE CASE
USING NAMESPACE
     resto-r=<http://chefmoz.org/rdf/elements/1.0#>,
     resto-d=<http://purl.org/dc/elements/1.0/>
```

2.3.4 Resultaten Samenvoegen

Werkwijze

De laatste stap is het samenvoegen van de resultaten. Dit onderdeel verschilt wat van de andere, omdat de ResultMerger het enige onderdeel is dat volledig vervangbaar is door een ander algoritme. Later meer hierover (3.2). Dit is niet de meest efficiënte implementatie van een join-algoritme. Bij het uitvoeren van configuratie-tests is hierdoor de invloed van de configuratie duidelijker merkbaar.

Alle resultaten zijn reeds verkregen, en het is nu een kwestie van de juiste tupels samen te voegen. De mappings die gebruikt worden voor een query, worden opgevraagd uit het systeem. Alle mappings worden vervolgens overlopen. Een mapping bevat informatie over de mappingvariabele en de identificatie van de repositories. Dit laatste wordt gebruikt om de resultaten van de subquery op te vragen.

Voor elk tupel van de eerste repository worden alle tupels van de tweede repository overlopen om te zien of de waarden van de mappingvariabele overeenkomen. Indien dit het geval is, mag men de tupels samenvoegen. Het zijn enkel de tupels die uiteindelijk samengevoegd worden die tot het eindresultaat behoren. Resultaten van subqueries die niet betrokken zijn in een mapping, moeten ook niet overlopen worden. Deze tupels worden wel teruggegeven in het eindresultaat.

Naar de gebruiker toe wordt een Hashtable teruggegeven. De key is de preprefix die bij een repository hoort en de value zijn de resultaten. Resultaten die samengevoegd zijn, worden bij beide preprefixen, die bij de mapping horen, in de tabel geplaatst. Zo kan de ontwikkelaar nog steeds zelf bepalen via welke key de informatie opgevraagd wordt. De berekening wordt slechts 1 maal gedaan, de enige overhead die dit met zich mee brengt is een extra toevoeging in de tabel.

Voorbeeld

Figuur 10 geeft weer hoe het resultaat gestructureerd is. De keys in dit geval zijn “cine” en “resto”, beide preprefixes gebruikt in het voorbeeld uit deze sectie. De values bij deze keys zijn dezelfde, namelijk het samengevoegd resultaat. Dit wordt gedupliceerd zodat de ontwikkelaar kan kiezen of deze resultaten via de “cine”-key of via die “resto-”key worden opgevraagd.

key	samengevoegde resultaten
cine	resto+cine
resto	resto+cine

Figuur 10: Hashtabel als resultaat

2.4 Besluit

Tijdens de ontwikkeling van DiSemDa zijn heel wat zaken afgewogen die de uitvoeringstijd beïnvloeden. Uit metingen blijkt dat het queries die door het systeem verstuurd worden een groot aandeel hebben in de totale uitvoeringstijd. Daarom dat er ook voor gekozen is dit aantal te minimaliseren.

DiSemDa is opgebouwd uit 3 grote modules. De eerste module zorgt ervoor dat een gedistribueerde query geanalyseerd wordt en opgesplitst wordt in subqueries. De tweede module voert deze subqueries uit via het Web met behulp van Sesame[9]. De volgorde van uitvoering van deze subqueries is van belang voor de uitvoeringstijd. Daarom dat de gebruiker de mogelijkheid heeft deze volgorde vast te leggen via de configuratie. De laatste module voegt de geschikte deelresultaten samen zodat hieraan niet merkbaar is dat het een gedistribueerde query betrof.

3

DiSemDa eigenschappen

DiSemDa is modulair opgebouwd. Het kan makkelijk aangepast worden naargelang de situatie waarin het gebruikt wordt. Natuurlijk is het belangrijk te weten waarop moet gelet worden bij de configuratie om de meest efficiënte keuze te maken. Dit wordt uitgelegd in dit hoofdstuk.

Verder worden er ook nog 2 voorbeelden gegeven waarvoor DiSemDa gebruikt kan worden. Alle stappen in het proces worden gevolgd: van configuratie over splitsen en uitvoeren tot het samenvoegen.

3.1 Optimaal Gebruik

Zoals reeds vermeld in het deel over de configuratie van DiSemDa(2.3.1), heeft de gebruiker de mogelijkheid de volledige volgorde van uitvoering te bepalen. De vraag stelt zich dan wat de meest efficiënte configuratie is. Dit wordt behandeld in het eerste deel van deze sectie. Er zijn ook andere factoren die de uitvoeringstijd beïnvloeden, zoals de snelheid van het netwerk, de eigenschappen van de servers en of er al dan niet variabelen in de "where"-clause aanwezig zijn. Deze factoren worden besproken in het de rest van deze sectie.

3.1.1 Configuratie

In het vorige hoofdstuk werd de volledige structuur van DiSemDa tot in de kleinste details uitgelegd. Het doel is natuurlijk de configuratie zo optimaal mogelijk te maken. Er moet rekening gehouden worden met de overload van het systeem. Met overload wordt data bedoeld die getransfereerd is, maar eigenlijk geen nut heeft, dit zijn dus de tupels die geen match hebben binnen een mapping.

Door de wijze waarop een query gestuurd wordt binnen DiSemDa, kan er enkel overload ontstaan in de resultaten van de eerste query van een mapping. De resultaten van de 2e repository van een mapping kunnen nooit last hebben van overload. De reden hiervoor is dat de query voor deze repository verrijkt is met de waarden van de mappingvariabele uit de eerste repository.

Dit heeft als gevolg dat, indien men de configuratie zo efficiënt mogelijk wilt opstellen, het nodig is dat de repository met het minst aantal waarden voor de mappingvariabele eerst wordt uitgevoerd binnen de mapping. Daarvoor moet de namespace van de mapping, waarvan deze repository gebruik maakt, als eerste namespace geplaatst worden in de mapping. Op deze manier is er het meest kans dat er zo weinig mogelijk overload is.

Dit is een zeer goede richtlijn, maar indien er meer informatie voorhanden is over de repositories in kwestie, is het natuurlijk mogelijk een beslissing te nemen waarvan de gevolgen duidelijker zijn. Het is dan nodig te weten hoeveel verschillende tupels geen match vinden binnen de mapping. De repository die de kleinste waarde voor dit getal heeft, wordt best als eerste geplaatst.

Hiermee rekening houden bevordert het verkeer over het netwerk: omdat er minder overload is, worden er ook minder onnodige tupels over het net verstuurd. Dit geeft echter op een andere plaats ook performantiewinst. De ResultMerger zal zijn werk sneller kunnen uitvoeren. Als er minder tupels zijn, gaat het samenvoegen natuurlijk ook sneller verlopen. Er zullen wel nog steeds evenveel tupels samengevoegd worden, maar er zijn minder tupels zonder match en dus zijn er minder acties nodig.

```
time split: 130 ms
mapping priority : 4
=> SELECT zipcode FROM {card} vcard:Pcode {zipcode} USING NAMESPAC
time query: 163 ms
=> SELECT DISTINCT cityname, cinename, zipcode FROM {cine} cine:Zi
time query: 9 ms
time execute : 176 ms
time merge 2 ms
total time : 309 ms
 final result
```

Figuur 11: Uitvoeringstijd configuratie vcard-cine

Deze stelling is getest aan de hand van 2 metingen. Het betreft een configuratie waarin de informatie van een vcard gelinkt wordt aan een bioscoop-repository. Figuur 11 geeft de informatie weer van een configuratie waar eerst de vcard-informatie opgehaald wordt en vervolgens de bioscoop-informatie. Figuur 12 is net omgekeerd. Er is een verschil van 69 milliseconden in de uitvoering in het voordeel van de eerste configuratie. In dit geval is het verschil niet zo groot, omdat het hier om kleine repositories gaat.

```
time split: 137 ms
mapping priority : 4
=> SELECT cityname, cinename, zipcode FROM {cine} cine:Zip {zipc
time query: 204 ms
=> SELECT DISTINCT zipcode FROM {card} vcard:Pcode {zipcode} WHE
time query: 30 ms
time execute : 238 ms
time merge 2 ms
total time : 378 ms
 final result
```

Figuur 12: Uitvoeringstijd configuratie cine-vcard

Waneer een mappingvariabele in verschillende mappings voorkomt, wordt het nog iets ingewikkelder. Er moet dan opgelet worden dat repositories die in vorige mappings voorkomen, als eerste repository in een mapping uitgevoerd worden. Aangezien resultaten van dergelijke repositories reeds bekend zijn van een vorige mapping, is het niet nodig nog een query te sturen naar deze repositories. De tweede repository binnen dezelfde mapping kan verrijkt worden met de informatie uit de vorige mappings. Dit zorgt er ook voor dat het resultaat van deze query selectiever zal zijn, wat de uitvoeringstijd bevordert. Stel dat de configuratie van een mapping in omgekeerde volgorde zou zijn: de eerste repository die uitgevoerd wordt heeft geen resultaten in vorige mappings, maar de tweede wel. Bij deze configuratie wordt de eerste query helemaal niet verrijkt. Dit zorgt voor een grotere overload.

3.1.2 Bandbreedte van het netwerk

DiSemDa blijft nog steeds een gedistribueerd systeem, waardoor er ook andere factoren dan de volgorde van uitvoering een rol spelen. Het is duidelijk dat de bandbreedte van het netwerk een rol speelt. Deze kan voor elke server anders zijn. Het is dan de bedoeling om zo weinig mogelijk data te transfereren over de tragere verbinding. Dit is echter afhankelijk van de grootte van de repository en de query die uitgevoerd wordt. In de context van deze studie kan er dus geen algemene uitspraak gedaan worden met betrekking tot de configuratie.

3.1.3 Variabelen in de “where”-clause

Indien er variabelen in de “where”-clause van een subquery staan, wordt deze subquery onmiddellijk heel wat selectiever. Dezelfde redenering als in de sectie over de configuratie (3.1.1) kan gemaakt worden: de repository geassocieerd met de selectievere query, wordt als eerste in de mapping geplaatst. Dan is er minder kans op overload. Maar deze selectiviteit moet steeds relatief tot de andere subquery binnen de mapping gezien worden. Daarom dat er geen vaste regel opgesteld kan worden.

3.1.4 Eigenschappen van de servers

Verschillende servers zullen ook vaak een verschillende uitvoeringstijd hebben voor een bepaalde query. Dit hangt van allerlei factoren af. Zo spelen de kloksnelheid en de load van de servers een belang. De verschillen in uitvoeringstijd kunnen vrij groot zijn, maar dit is niet enkel afhankelijk van voorgenoemde factoren. Het verschil is een combinatie van eigenschappen van de query in kwestie en van de implementatie van Sesame[9].

3.2 DiSemDa Doeleinden

Modulariteit was van bij de start van het ontwikkelingsproces van DiSemDa al een punt waar veel aandacht aan besteed werd. DiSemDa in zijn huidige vorm is een systeem dat verbanden legt tussen repositories, verbanden die op een gelijkheid van waarden steunen (gelijke zipcodes in voorbeeld 2). Om deze gelijkheid te abstraheren, is er gebruik gemaakt van de interfaces SplitterConfig, QueryEnricher en ResultMerger. Het is dus mogelijk deze gelijkheid te vervangen door bv. een kleiner dan: prijzen van een product uit repository 1 die kleiner zijn dan een product uit repository 2.

De SplitterConfig berekent de mappingvariabele uit een query. Het is in te denken dat wanneer er niet op gelijkheid gemapped wordt, de mappingvariabele op een andere manier berekend moet worden. Daarom dat er abstractie van dit onderdeel uit de QuerySplitter gemaakt is.

De QueryEnricher speelt ook een belangrijke rol. Indien men weer het voorbeeld beschouwt uit de eerste alinea van deze subsectie, is het duidelijk dat de query op een andere manier verrijkt moet worden. In de huidige configuratie wordt de "where"-clause uitgebreid met gelijkheids-operatoren. Door dit te abstraheren is het mogelijk ook op andere manieren een query te verrijken.

Om net dezelfde reden is de ResultMerger geabstraheerd. In DiSemDa zoals deze in dit document voorgesteld is, worden tupels met dezelfde waarden voor de mappingvariabele samengevoegd. Dit kan dus ook op andere criteria (kleiner dan, groter dan) in andere situaties.

De abstractie van bovenstaande elementen laat toe een systeem te configureren met een heel ander gedrag dan het huidige. DiSemDa is dus niet beperkt in de vorm die hier uitgelegd wordt in dit document, maar er is plaats voor nog tal van uitbreidingen en andere manieren van gebruik.

3.3 Toepassingen

3.3.1 Verrijking

Een voorbeeld van een toepassing van DiSemDa is het verrijken van data van de ene repository met informatie uit een andere repository. Het voorbeeld dat hiervoor gebruikt wordt, verbindt een bioscoop-repository met een restaurant-repository. De link tussen beide is de postcode waarin een bioscoop of restaurant zich bevindt.

Voorbeeld 11

```
restaurant repository:
preprefix:              resto
Sesame URL:             http://serverke.dyndns:8080/openrdf
Repository name:        resto

restaurant repository:
preprefix:              cine
Sesame URL:             http://demoserver.org:8001/openrdf
Repository name:        cine
```

Voorbeeld 11 geeft de configuratie van de repositories die in deze opstelling gebruikt is. Beide repositories worden op dat moment toegevoegd aan het systeem. Vervolgens is het nodig de mapping te definiëren. Dit gebeurt in voorbeeld 12. De eerste namespace wordt gebruikt in de bioscoop-repository, de tweede namespace in de restaurant-repository. De predikaten worden ook opgegeven, nl. "Zip" in beide gevallen. De prioriteit is hier niet van belang aangezien er slechts één mapping opgegeven is.

Voorbeeld 12

```
namespace 1:            http://www.vub.ac.be/cinema#
predikaat 1:            Zip
namespace 2:            http://chefmoz.org/rdf/elements/1.0#
predikaat 2:            Zip
prioriteit :            1
```

Deze configuratie laat een samenspel van restaurants en bioscopen toe. Om een mapping te gebruiken is dit echter niet genoeg. Deze mapping moet ook terug te vinden zijn in de query. Dergelijke query wordt weergegeven in voorbeeld 13. Deze query zorgt ervoor dat koppels opgebouwd worden tussen restaurants en bioscopen die zich in dezelfde postcode bevinden. De variabele "zipcode" wordt in de "from"-clause in beide repositories gebruikt, dit stelt de mappingvariabele voor.

Voorbeeld 13

```
SELECT  zipcode, cinename, cityname, restoname
FROM  {cine} cine:City {cityname},
      {cine} cine:Name {cinename},
      {cine} cine:Zip {zipcode},
      {resto} resto-d:Title {restoname},
      {resto} resto-r:Cuisine {cuisinetype},
      {resto} resto-r:Zip {zipcode}
USING NAMESPACE
      cine = <http://www.vub.ac.be/cinema#>,
      resto-r = <http://chefmoz.org/rdf/elements/1.0#>,
      resto-d = <http://purl.org/dc/elements/1.0/>
```

In de eerste stap wordt de query uit voorbeeld 13 opgesplitst in 2 subqueries. Eerst wordt de query op de bioscoop-repository uitgevoerd (voorbeeld 14). De verschillende waarden van de zipcodes worden uit dit resultaat gehaald en hiermee wordt de query voor de restaurant-repository verrijkt. Dit is in voorbeeld 15 aangetoond door de waarden in de "where"-clause.

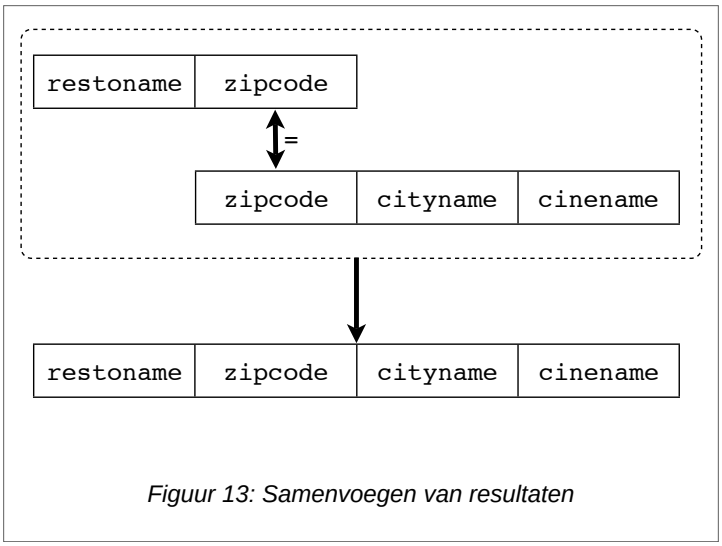
Voorbeeld 14

```
SELECT cityname, cinename, zipcode
FROM  {cine} cine:City {cityname},
      {cine} cine:Name {cinename},
      {cine} cine:Zip {zipcode}
USING NAMESPACE
      cine = <http://www.vub.ac.be/cinema#>
```

Wanneer beide queries uitgevoerd zijn, is de tijd gekomen om de resultaten samen te voegen. De mappingvariabele is in dit geval "zipcode". Wanneer er tupels zijn met dezelfde zipcode, worden deze samengevoegd en bij het eindresultaat gevoegd (figuur 13). Dit eindresultaat is een hashtabel(zie figuur 10) waarvan de tupels, die bij beide repositories horen, van de vorm zijn die onderaan in figuur 10 wordt weergegeven. Het eindresultaat bevat meer dan 1000 tupels.

Voorbeeld 15

```
SELECT DISTINCT restoname, zipcode
FROM  {resto} resto-d:Title {restoname},
      {resto} resto-r:Cuisine {cuisinetype},
      {resto} resto-r:Zip {zipcode}
WHERE zipcode LIKE "8200" IGNORE CASE OR
     zipcode LIKE "1020" IGNORE CASE OR
     zipcode LIKE "9000" IGNORE CASE OR
     zipcode LIKE "3500" IGNORE CASE OR
     zipcode LIKE "8300" IGNORE CASE OR
     zipcode LIKE "8000" IGNORE CASE OR
     zipcode LIKE "8400" IGNORE CASE OR
     zipcode LIKE "1420" IGNORE CASE OR
     zipcode LIKE "1000" IGNORE CASE OR
     zipcode LIKE "8500" IGNORE CASE OR
     zipcode LIKE "8670" IGNORE CASE
USING NAMESPACE
      resto-r = <http://chefmoz.org/rdf/elements/1.0#>,
      resto-d = <http://purl.org/dc/elements/1.0/>
```



Figuur 13: Samenvoegen van resultaten

3.3.2 Personalisatie

Het is mogelijk DiSemDa te gebruiken voor personalisatie. In dit voorbeeld maken we gebruik van de bioscoop- en restaurant-repository van in het vorige voorbeeld, maar ook van een vcard-repository. Een vcard houdt persoonlijke informatie bij over een gebruiker, in ons geval in RDF/XML formaat [10]. Deze informatie kan bv. opgeslagen worden in een gebruikersprofiel van een applicatie of in een browser. Deze laatste aanpak laat toe dat ook andere applicaties van dezelfde informatie gebruik kunnen maken om zichzelf te personaliseren.

Deze toepassing zorgt er voor dat er allereerst de postcode van de vcard wordt opgehaald. Deze informatie wordt gebruikt om de bioscoop-informatie te personaliseren: enkel de bioscopen die zich in dezelfde postcode bevinden, zijn van belang. Vervolgens worden ook nog de Italiaanse restaurants in rekening gebracht. Dit voorbeeld is een gepersonaliseerde versie van het voorbeeld uit 3.3.1. Alle bioscopen met een Italiaans restaurant in dezelfde postcode worden opgevraagd. Enkel worden nu de resultaten beperkt tot de zipcode uit de gebruikersgegevens.

Voorbeeld 16

vcard repository:

```
preprefix:              vcard
Sesame URL:             http://localhost:8080/openrdf
Repository name:        default
```

Voorbeeld 17

```
namespace 1:        http://www.w3.org/2001/vcard-rdf/3.0#
predikaat 1:        Pcode
namespace 2:        http://www.vub.ac.be/cinema#
predikaat 2:        Zip
prioriteit :        4
```

De configuratie houdt in dat alle repositories toegevoegd worden en vervolgens de mapping aangemaakt wordt. Er wordt van uit gegaan dat de configuratie uit het vorig voorbeeld nog steeds in het systeem zit (voorbeeld 11 en 12). Voorbeeld 16 voegt dan de nieuwe vcard-repository toe. Voorbeeld 17 geeft de nieuwe mapping weer die toegevoegd wordt aan het systeem. Dit is een voorbeeld waarin de namen van de predikaten die aan elkaar gelinked worden niet overeenkomen. De prioriteit van de nieuwe mapping is hoger, nl *4*, en deze zal dus eerst uitgevoerd worden. Merk op dat de informatie uit de vcard niet onmiddellijk aan de restaurant-informatie gelinked wordt, maar enkel aan de bioscoop-informatie.

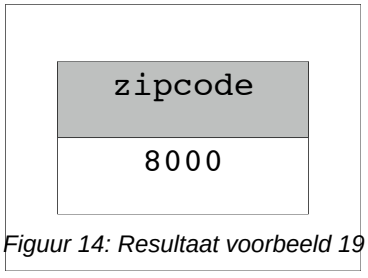
Voorbeeld 18

```
SELECT  zipcode, cinename, cityname, restoname
FROM  {cine} cine:City {cityname},
      {cine} cine:Name {cinename},
      {cine} cine:Zip {zipcode},
      {resto} resto-d:Title {restoname},
      {resto} resto-r:Cuisine {cuisinetype},
      {resto} resto-r:Zip {zipcode},
      {card} vcard:Pcode {zipcode}
WHERE cuisinetype LIKE “Italian”
USING NAMESPACE
      cine = <http://www.vub.ac.be/cinema#>,
      resto-r = <http://chefmoz.org/rdf/elements/1.0#>,
      resto-d = <http://purl.org/dc/elements/1.0/>,
      vcard = <http://www.w3.org/2001/vcard-rdf/3.0#>
```

De query in voorbeeld 18 wordt opgesplitst. Door de hogere prioriteit van de laatst toegevoegde mapping wordt de vcard repository eerst gequeried. Dit gebeurt met de query uit voorbeeld 19. Deze query haalt de postcode op uit het profiel van de gebruiker (figuur 14). Vervolgens wordt de query voor de bioscoop-repository verrijkt met deze ene waarde. De resulterende query wordt weergegeven in voorbeeld 20. Er komt 1 resultaat terug, namelijk Cinéma Lumière (figuur 15). Het is hier duidelijk dat er minder resultaten terugkomen dan in het vorige voorbeeld. Dit komt de performantie ten goede, zowel in de volgende mapping als bij het samenvoegen.

Voorbeeld 19

```
SELECT zipcode
FROM  {card} vcard:Pcode {zipcode}
USING NAMESPACE
      vcard = <http://www.w3.org/2001/vcard-rdf/3.0#>
```



Figuur 14: Resultaat voorbeeld 19

Voorbeeld 20

```
SELECT DISTINCT cityname, cinename, zipcode
FROM  {cine} cine:Zip {zipcode},
      {cine} cine:Name {cinename},
      {cine} cine:City {cityname}
WHERE zipcode LIKE "8000" IGNORE CASE
USING NAMESPACE cine = <http://www.vub.ac.be/cinema#>
```

cityname	cinename	zipcode
Brugge	Cinéma Lumière	8000

Figuur 15: Resultaat voorbeeld 20

Op dit moment in de uitvoering, is de eerste mapping afgewerkt. Dan wordt de volgende mapping, tussen de bioscoop-repository en de restaurant-repository, behandeld. Normaal moet de bioscoop-repository gequeried worden, maar aangezien deze resultaten reeds bekend zijn, is het zinloos van dit werk nog eens uit te voeren. De volgende stap is een query sturen naar de restaurant-repository. Deze wordt verrijkt met de resultaten uit de vorige mapping van de bioscoop-repository. Dit wordt getoond in voorbeeld 21.

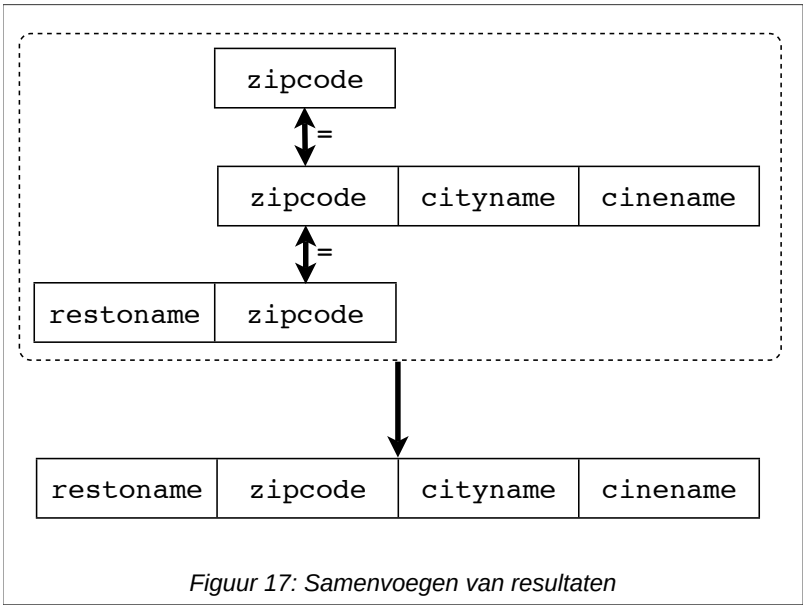
Voorbeeld 21

```
SELECT DISTINCT restoname, zipcode
FROM  {resto} resto-d:Title {restoname},
      {resto} resto-r:Zip {zipcode},
      {resto} resto-r:Cuisine {cuisinetype}
WHERE     cuisinetype LIKE "Italian"
      AND (zipcode LIKE "8000" IGNORE CASE)
USING NAMESPACE
      resto-d = <http://purl.org/dc/elements/1.0/>,
      resto-r = <http://chefmoz.org/rdf/elements/1.0#>
```

cityname	cinename
La Torre	8000
La Due Venezie	8000
Loreto	8000
Pastis	8000
Relais Ravestein	8000
Riva Del Sole	8000
Trium	8000
Trium Tattoria	8000

Figuur 16: Resultaat voorbeeld 21

Het is duidelijk dat deze query veel selectiever is dan dezelfde query uit het vorige voorbeeld (voorbeeld 15). Dit omdat deze onrechtstreeks verrijkt is met de informatie uit de vcard. De resultaten van deze query zijn terug te vinden in figuur 16.



Figuur 17: Samenvoegen van resultaten

De laatste stap is het samenvoegen van de verschillende deelresultaten (figuur 17). Dit komt op hetzelfde neer als in het eerste voorbeeld, enkel zijn er veel minder tupels die gemerged moeten worden. Dit komt doordat de informatie uit de vcard toelaat veel selectiever informatie op te halen. In dit geval zullen alle keys in het eindresultaat geassocieerd worden met dezelfde waardentabel. Dit is een gevolg van het feit dat beide mappings van dezelfde mappingvariabele gebruik maken. Zo een waardentabel wordt teruggegeven in figuur 18.

restoname	zipcode	cityname	cinename
La Torre	8000	Brugge	Cinéma Lumière
La Due Venezie	8000	Brugge	Cinéma Lumière
Loreto	8000	Brugge	Cinéma Lumière
Pastis	8000	Brugge	Cinéma Lumière
Relais Ravestein	8000	Brugge	Cinéma Lumière
Riva Del Sole	8000	Brugge	Cinéma Lumière
Trium	8000	Brugge	Cinéma Lumière
Trium Tattoria	8000	Brugge	Cinéma Lumière

Figuur 18: Eindresultaat personalisatie

4

Nabeschouwing

DiSemDa is geëvolueerd tot een dynamisch, gedistribueerd querysysteem dat in staat stelt met verschillende configuraties voor distributie te experimenteren. In dit hoofdstuk wordt aangetoond dat DiSemDa een meerwaarde biedt aan eindgebruiker en ontwikkelaar. In de laatste sectie worden enerzijds enkele zaken aangehaald die nog niet ondersteund worden en anderzijds boeiende uitbreidingen mogelijk op vlak van functionaliteit.

4.1 Conclusie

DiSemDa laat toe queries uit te voeren op gedistribueerde bronnen. Aan de reeds bestaande applicaties is slechts minimale aanpassing nodig: er moet namelijk een instantie van Sesame Server beschikbaar gesteld worden. DiSemDa gaat ook verder dan andere mapping frameworks. Het is een volledige oplossing dat als package gebruikt kan worden door een ontwikkelaar.

Het DiSemDa-systeem biedt een meerwaarde aan de ontwikkelaar: repositories kunnen gecombineerd worden zonder zelf een lokale kopie te moeten onderhouden. Ook wordt er een heel dynamisch systeem aangeboden wat toelaat meerdere applicaties op dezelfde DiSemDa-instantie te laten lopen.

De meerwaarde voor gebruikers van systemen die ontwikkeld zijn met behulp van DiSemDa, is de rijkere informatie die aangeboden wordt. Relaties tussen verschillende repositories kunnen ten volle uitgebuit worden in het systeem. Dit is iets wat de gebruiker anders zelf moest doen, zoals bvb restaurants opzoeken in de buurt van een bioscoop.

DiSemDa is ontwikkeld in de geest van het Semantisch Web. Door de data niet te kopiëren worden update-problemen vermeden, maar wordt ook toegelaten dat verschillende DiSemDa-instanties dezelfde repositories kunnen opnemen in de configuratie. Dit is uitermate geschikt voor gebruik op het Web. Data moet niet meer gedupliceerd worden om het op te nemen in een applicatie. Zo zal ook elke applicatie steeds over de laatst aangepaste versie van de dataset beschikken.

De menselijke interactie die nodig is voor de ontwikkelaar om een gedistribueerde query uit te voeren is echt minimaal: er is een configuratie nodig van de repositories en de verbinding tussen de repositories. Er is dus wel enige kennis nodig over de ontologieën om een juiste verbinding tot stand te kunnen brengen.

DiSemDa is een werkend systeem dat voldoet aan de verwachtingen die vooropgesteld waren. Er is nog heel wat ruimte voor uitbreidingen en aanpassingen. Dynamiek was van in het begin zeer belangrijk, naast het uitspelen van relaties tussen repositories. Dit zijn ook net de belangrijkste troeven van DiSemDa.

4.2 Toekomstig Werk

4.2.1 Modulariteit

Door het feit dat DiSemDa heel dynamisch opgebouwd is, zijn er nog heel wat uitbreidingen mogelijk. De modulariteit van het programma is reeds aan bod gekomen. Deze eigenschap houdt in dat de door het schrijven van een nieuwe SplitterConfig, QueryEnricher en ResultMerger, het systeem zich volledig anders zou gedragen. Om DiSemDa nog dynamischer te maken, zou het aangewezen zijn om een uitbreiding toe te voegen, zodat per mapping een andere combinatie van een SplitterConfig, QueryEnricher en ResultMerger gespecificeerd kan worden. Dit zou mogelijk maken dat elke mapping op een andere manier kan reageren terwijl alles binnen eenzelfde instantie van DiSemDa gebeurt.

Voorbeeld 22

```
WHERE cuisinetype IN ( SELECT cuisine
                       FROM {rest} dummy:type {cuisine}
                       USING NAMESPACE
                            dummy = <www.dummy.org/rdf#> )
```

4.2.2 Functionaliteit

Het deel waar queries gesplitst worden, kan ook nog uitgebreid worden. Zo worden op dit moment enkel normale, simpele queries, zoals deze in de voorbeelden getoond zijn, ondersteund. De constructie waar in de where clause een subquery gebruikt wordt (voorbeeld 22) wordt nog niet ondersteund. De vraag is in welke mate deze constructie gebruikt zal worden. Hetzelfde resultaat kan bekomen worden door dit te specificeren in de "from"-clause van de originele query. Dit wordt getoond in voorbeeld 23.

Voorbeeld 23

```
FROM  {resto} resto:cuisinetype {cuisinetype},
      {rest} dummy:type {cuisinetype}
USING NAMESPACE
      resto = <http://chefmoz.org/rdf/elements/1.0#>,
      dummy = <www.dummy.org/rdf#>
```

Ook wordt een "*" en "distinct" in de "select"-clause van de originele query nog niet ondersteund omdat deze elementen niet herkend worden bij het inlezen van een query. Om dezelfde reden worden volgende primitieven, ingebouwd in Sesame, niet ondersteund: union, minus en intersect.

In de huidige versie van DiSemDa is het verplicht de mappingvariabele in de "select"-clause van de originele query te vermelden. Anders wordt de waarde van de mappingvariabele niet opgevraagd in de subqueries en kunnen er vervolgens geen tupels samengevoegd worden op basis van deze mappingvariabele. Een uitbreiding kan ervoor zorgen dat zelf als de mappingvariabele niet vermeld wordt in de originele query, deze toch in de "select"-clause van de subqueries toegevoegd wordt.

Een andere uitbreiding, die ook de configuratie ten goede zou komen, is een uitbreiding waardoor DiSemDa zelf de volgorde van uitvoering van de repositories binnen één mapping bepaalt. In sectie 3.1 worden de verschillende variabelen gegeven die mee spelen in de keuze van de volgorde. Dit zou kunnen verwerkt worden binnen DiSemDa in een algoritme. Voor het beste resultaat te verkrijgen, zou het interessant zijn om dit te combineren met een learning-algoritme.

DiSemDa maakt wel gebruik van Sesame, maar toch is het daar volledig onafhankelijk van. Het is mogelijk om een andere query engine te gebruiken. De enige vereiste is dat deze query engine een SeRQL-query kan uitvoeren via het net op een bepaalde server. Dit is wel belangrijk om nieuwe query engines, zoals bijvoorbeeld vermeld in [21], te kunnen gebruiken.

4.2.3 Performantie

Het samenvoegen van resultaten is de laatste stap in het uitvoeringsproces, maar hier kan nog wat aan verbeterd worden. De ResultMerger die nu gebruikt wordt in DiSemDa is heel recht voor de raap. Er is geen geavanceerd join-algoritme gebruikt. Er kan dus nog performantiewinst geboekt worden in deze module. Dit aanpassen zal op zich geen ingewikkelde taak zijn. Een ResultMerger is immers een onderdeel van de configuratie, en DiSemDa is erop voorzien om makkelijk een andere configuratie toe te laten.

Qua performantie zou er nog met iets anders rekening gehouden kunnen worden. Indien er meerdere mappings op dezelfde mappingvariabele gedefinieerd worden, is het aangewezen met de volgorde van uitvoering rekening te houden (3.1.1).Nu ligt dit in de handen van de gebruiker, het zou handig zijn mocht DiSemDa autonoom kunnen bepalen wat de meest geschikte volgorde is: repositories die al gequeried zijn, worden best eerst uitgevoerd binnen een mapping.

Bronnen

[1] Internet History, http://www.computerhistory.org/exhibits/internet_history/

[2] Cristian Pérez de Laborda and Stefan Conrad, *Database to Semantic Web Mapping Using RDF Query Languages*, Institute of Computer Science, Heinrich-Heine-Universität Düsseldorf, Germany — Springer Berlin / Heidelberg, 2006, ISBN 978-3-540-47224-7

[3] Cristian Pérez de Laborda and Stefan Conrad, *Relational.OWL, A Data and Schema Representation Format Based on OWL*, Institute of Computer Science, Heinrich-Heine-Universität Düsseldorf, Germany — Australian Computer Society, 2005, ISBN 1-920682-25-2

[4] Namyoun Choi, Il-Yeol Sing and Hyoil Han, *A Survey on Ontology Mapping*, College of Information Science and Technology, Drexel University Philadelphia — ACM Press, 2006, ISSN 0163-5808

[5] Alexander Maedche, Boris Motik, Nuno Silva, Raphael Volz, *MAFRA — A MApping FRAmework for Distributed Ontologies*, Forschungszentrum informatik, Univ. Karlsruhe, Germany — Springer Verlag, 2002, ISSN 0302-9743

[6] DAML, The DARPA Agent Markup Language, http://www.daml.org/

[7] Vaida Jakoniene, Ontology Integration, http://www.ida.liu.se/labs/iislab/courses/LW/slides/ontologyIntegration.pdf

[8] Yannis Kalfoglou and Marco Schorlemmer, *Ontology Mapping: The State of the Art*, Department of Electronics and Computer Science, University of Southampton, England, UK and Centre for Intelligent Systems and their Applications, University of Edingburgh, Scotland, UK — The Knowledge Engineering Review, 2003, ISSN 0269-8889

[9] *Sesame, an open source RDF framework*, http://www.openrdf.org/

[10] *Representing Vcard objects in RDF/Xml*, http://www.w3.org/TR/vcard-rdf

[11] *The SeRQL Query Language*, http://www.openrdf.org/doc/sesame/users/ch06.html

[12] *SPARQL, a Query Language for RDF*, http://www.w3.org/TR/rdf-sparql-query/

[13] *Resource Description Framework (RDF)*, http://www.w3.org/RDF/

[14] M. Goossens, Cursus *Gedistribueerde Systemen*, 2005-2006

[15] *RDF/XML Syntax Specification*, http://www.w3.org/TR/rdf-syntax-grammar/

[16] *Turtle — Terse RDF Triple Language*, http://www.dajobe.org/2004/01/turtle/

[17] *Notation 3 — A Readable Language For Data on the Web*, http://www.w3.org/DesignIssues/Notation3

[18] *N-Triples*, http://www.w3.org/TR/rdf-testcases/#ntriples

[19] *TRIX — RDF triples in XML*, http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Stickler01/EML2004Stickler01.html

[20] *Apache Tomcat*, http://tomcat.apache.org/

[21] *Super-fast RDF Search Engine Developed*, http://www.theregister.co.uk/2007/05/04/semantic_web_breakthrough/

Tabel der Figuren

Dankwoord

Mijn oprechte appreciatie en dank gaan uit naar Prof. Dr. Ir. Geert-Jan Houben en Dr. Sven Casteleyn voor de goede begeleiding gedurende het hele proces dat achter deze thesis verborgen is.

Ook wil ik Mathieu Cardinael bedanken voor de goede samenwerking gedurende het denk- en ontwikkelingsproces.