



Vrije Universiteit Brussel

Faculty of Science
Department of Computer science

Realizing a Query Pipeline and Achieving Query Caching in the Mobile Semantic Web

Graduation thesis submitted in partial fulfillment of the requirements for the degree of
Master in Applied Computer Science.

Stijn Vlaes

Promotor: Prof. Dr. Olga De Troyer

Advisor: William Van Woensel

Academic year 2010-2011



Acknowledgements

Using these acknowledgements I would like to thank everyone that helped me achieve thesis. First of all I would like to thank my promoter, Prof. Olga De Troyer, for giving me the chance to realize this thesis.

Secondly, I would like to express my deepest gratitude to my advisor, PhD student, William Van Woensel. I am thankful for all the personal assistance, guidance, moral support and all the knowledge that was shared with me.

I also want to thanks my classmate, Raf Walravens, for all the help he provided during the course of this thesis.

I would also like to thank Tania Van Denhouwe, for her moral support and encouragement whenever I needed it during the course of this thesis.

Finally I would like to thank my family for their love and support during my entire life and for supporting me getting my master's degree and achieving this thesis.

Abstract

Nowadays mobile devices (e.g. smart phone, PDA) have become part of everyday life. Not only do they allow people to make phone calls or send text messages; they also allow people to access the Web using a dedicated mobile browser. At the same time, the hardware limitations these devices used to have are fading (e.g. faster processors, more memory and improved connectivity) and new mobile-specific hardware capabilities are being added (e.g. GPS, RFID). This evolution allows mobile developers to create bigger and better applications, which are able to map the user's environment (e.g. using RFID to detect nearby physical entities). When combining these improvements with access to the Web, mobile applications can provide relevant information about the user's surroundings, retrieved from services and nearby objects. For instance, shops that are nearby that sell a certain product the user needs, without the user having to browse the Web for this information. Moreover, applications can automatically notify the user whenever something the user might be interested in is around.

The SCOUT framework, which is currently being developed at the WISE lab of the VUB, supports the development of context-aware mobile applications. These applications are able to offer relevant information and services to the user, based on the user's environment and needs at a given time and place. The SCOUT framework achieves this by providing a conceptual and integrated view on the environment called the Environment Model, which is comprised of metadata on the physical entities found nearby the user, together with the user's profile information. The SCOUT framework also allows developers to provide feedback to a user whenever a new source (e.g. a shop) is detected that provides something the user might be interested in (e.g. a shop which sells shoes the user encountered in another store in the past week) by making use of the Notification Service.

Until now, whenever such notifications had to be checked, the Notification Service had to execute the associated query on both the new source and the Environment Model, making this a costly process. This thesis investigates how a Notification Query service can be managed more efficiently. For this purpose, two mechanisms have been investigated and extensively tested.

First, we have investigated a query pipeline mechanism, which extracts two different parts from given Notification Service queries (i.e. the part of the query that has to be executed on the new source, and the part of the query that has to be executed on the Environment Model). To achieve this, two strategies have been investigated which differ in the way the second part is constructed and executed on the Environment Model.

Secondly, we have developed a query caching mechanism, which is able to store results for a given query, which later on can be re-used if a similar query is executed on the Environment Model. Allowing this system to not only be used by the Notification Service, but also by the Query Service.

Abstract (Dutch)

Tegenwoordig maken mobiele apparaten (vb. smart phone, PDA) deel uit van het dagelijkse leven. Deze apparaten laten mensen niet alleen toe om te telefoneren of tekst berichten te versturen, ze geven mensen ook toegang tot het Web door middel van een toegewijde mobiele browser. Tegelijkertijd zijn de beperkingen op de hardware aan het verdwijnen (vb. snellere processors, meer geheugen en verbeterde connectiviteit), alsook wordt mobiel-specifieke hardware toegevoegd (vb. GPS, RFID). Deze evolutie laat mobile developers toe om grotere en betere applicaties te ontwikkelen, die de gebruikersomgeving in kaart kunnen brengen (vb. RFID gebruiken om nabij gelegen fysieke entiteiten te detecteren). Wanneer deze verbeteringen gecombineerd worden met de toegang tot het Web, kunnen mobiele applicaties de relevante informatie over de gebruikersomgeving verstrekken die ze verkregen van diensten en nabije objecten. Bijvoorbeeld, informatie over nabij gelegen winkels die een bepaald product verkopen welke de gebruiker nodig heeft, kan verschaft worden zonder dat de gebruiker op het Web moet surfen. Bovendien kunnen deze applicaties de gebruiker automatisch verwittigen wanneer men iets belangrijk kan aantreffen in de omgeving.

Het SCOUT framework, dat momenteel wordt ontwikkeld in het WISE lab van de VUB, ondersteunt de ontwikkeling van context-aware mobiele applicaties. Deze applicaties laten toe om aan de gebruiker relevante informatie en diensten aan te bieden die gebaseerd zijn op de gebruikersomgeving en de behoeften op een bepaald moment en plaats. Het SCOUT framework bereikt dit door een conceptuele en geïntegreerde aanblik op de omgeving aan te bieden, namelijk het Environment Model. Dit Model omvat metadata over de informatie in het profiel van de gebruiker en de fysieke entiteiten die in de buurt van de gebruiker kunnen gevonden worden. Het SCOUT framework laat ontwikkelaars ook toe om aan de hand van de Notification service, de gebruiker feedback te geven wanneer een nieuwe bron (vb. een winkel) wordt ontdekt die iets aanbiedt waar de gebruiker geïnteresseerd in is (vb. een winkel waar schoenen worden verkocht die de gebruiker de voorbije week in een andere winkel aantrof).

Tot nu toe moest de Notification Service, bij het controlleren van aankondigingen, de geassocieerde query uitvoeren op de nieuwe bron en het Environment Model, waardoor dit een duur proces was. Deze thesis onderzoekt hoe een Notifications Query service meer efficiënt kan bestuurd worden. Hiervoor werden twee mechanismen onderzocht en uitgebreid getest.

Eerst werd een query pipeline mechanisme onderzocht, die twee verschillende delen afleidt van gegeven Notification Service Queries (i.e. het deel van de query dat moet uitgevoerd worden op de nieuwe bron en het deel van de query dat moet uitgevoerd worden op het Environment Model). Om dit te bereiken werden twee strategieën onderzocht die verschillen in de manier waarop de tweede sub query word geconstrueerd en uitgevoerd.

Verder werd een query caching mechanisme ontwikkeld die resultaten voor een gegeven query kan opslaan zodat deze resultaten later opnieuw kunnen gebruikt worden als een gelijkaardige query op het Environmental Model wordt uitgevoerd, waardoor niet alleen de Notification Service deze resultaten kan gebruiken, maar ook de Query service.

Table of Contents

Chapter 1.	Introduction.....	- 1 -
1.1	Context	- 1 -
1.2	Problem Description.....	- 1 -
1.3	Approach	- 2 -
1.4	Thesis Structure.....	- 2 -
Chapter 2.	Background.....	- 3 -
2.1	SCOUT framework	- 3 -
2.1.1	Introduction.....	- 3 -
2.1.2	Overview.....	- 3 -
2.1.3	Detection Layer	- 4 -
2.2	Android.....	- 7 -
2.3	Semantic Web	- 8 -
2.3.1	The idea and purpose of the Semantic Web.....	- 8 -
2.3.2	The Semantic Web Stack	- 9 -
2.4	Resource Description Framework	- 10 -
2.4.1	What	- 10 -
2.4.2	Representation	- 10 -
2.5	RDF Schema.....	- 11 -
2.6	Web Ontology Language	- 12 -
2.7	SPARQL.....	- 13 -
2.7.1	Introduction.....	- 13 -
2.7.2	Syntax	- 13 -
2.8	Semantic Web Frameworks.....	- 16 -
2.9	Caching	- 16 -
2.9.1	Introduction.....	- 16 -
2.9.2	For what data	- 16 -
2.9.3	How does it work.....	- 17 -
2.9.4	Query caching.....	- 17 -
Chapter 3.	Related Work.....	- 18 -
3.1	The SCOUT framework	- 18 -
3.1.1	Linking of physical entities to web presences	- 18 -
3.1.2	Location specific data storage	- 19 -
3.1.3	Integrated view of context sources.....	- 20 -

3.1.4	Determining relevant sources for a given query	- 21 -
3.2	Query Pipelining	- 21 -
3.3	Caching mechanisms	- 22 -
3.3.1	Client side caching mechanisms	- 22 -
3.3.2	Caching query results	- 23 -
3.3.3	Cache invalidation techniques.....	- 25 -
3.3.4	Cache Replacement policies	- 26 -
Chapter 4.	Query Pipelining	- 28 -
4.1	Query analysis	- 28 -
4.1.1	Automatic query analysis	- 29 -
4.1.2	Query analysis using named graphs	- 30 -
4.2	Constructing the queries	- 30 -
4.2.1	Join strategy	- 31 -
4.2.2	Injection strategy.....	- 31 -
4.3	Joining the results.....	- 32 -
Chapter 5.	Cache	- 34 -
5.1	Query caching.....	- 34 -
5.2	Query graph construction	- 34 -
5.3	Query graph comparison.....	- 35 -
5.3.1	Graph traversal.....	- 36 -
5.4	Eviction Strategy.....	- 38 -
5.4.1	Add an element to the cache	- 39 -
5.4.2	Update an element in the cache	- 39 -
5.4.3	Find an element in the cache	- 40 -
Chapter 6.	Implementation.....	- 41 -
6.1	Overview.....	- 41 -
6.1.1	Integration in the SCOUT framework	- 41 -
6.1.2	Encountering a new source	- 42 -
6.1.3	Executing a query	- 42 -
6.2	Query Pipeline	- 43 -
6.2.1	Splitting up the query	- 43 -
6.2.2	Query Construction and execution	- 45 -
6.2.3	Joining the results.....	- 47 -
6.3	Comparing queries	- 48 -

6.3.1	Building query graphs.....	- 48 -
6.3.2	Comparing query graphs	- 52 -
6.4	Caching strategy	- 55 -
6.4.1	Least Recently Used.....	- 57 -
6.4.2	Cache Manager.....	- 58 -
Chapter 7.	Evaluation.....	- 59 -
7.1	Testing environment	- 59 -
7.2	Query Pipeline	- 59 -
7.2.1	Test cases.....	- 59 -
7.2.2	Criterion 1: Analysis time	- 67 -
7.2.3	Criterion 2: Construction time.....	- 67 -
7.2.4	Criterion 3: Sub query execution time	- 68 -
7.2.5	Criterion 4: Join time	- 69 -
7.2.6	Criterion 5: Execution time of the original query compared to the execution using the query pipeline.....	- 70 -
7.2.7	Criterion 6: Execution time of the entire test	- 70 -
7.2.8	Criterion 7: Amount of results for each query using the two strategies.....	- 71 -
7.2.9	Conclusion	- 72 -
7.3	Cache.....	- 72 -
7.3.1	Test cases.....	- 72 -
7.3.2	Criterion 1: Construction time.....	- 73 -
7.3.3	Criterion 2: Comparison time	- 73 -
7.3.4	Criterion 3: Execution time of the original query	- 74 -
7.3.5	Criterion 4: Execution time of the entire test	- 75 -
7.3.6	Criterion 5: Amount of time needed to add new sources.....	- 75 -
7.3.7	Conclusion	- 76 -
Chapter 8.	Conclusion	- 77 -
8.1	Summary.....	- 77 -
8.1.1	Query Pipeline	- 77 -
8.1.2	Cache strategy	- 77 -
8.2	Future work	- 78 -
8.2.1	Query pipeline	- 78 -
8.2.2	Query caching.....	- 78 -
Chapter 9.	Bibliography.....	- 80 -

List of charts

Chart 1. Querypart analys time - 67 -
Chart 2. Querypart construction time - 68 -
Chart 3. Sub query execution time..... - 69 -
Chart 4. Sub query join time..... - 69 -
Chart 5. Execution time of original query (compared to pipeline) - 70 -
Chart 6. Execution time of each test - 71 -
Chart 7. Amount of results using each pipeline strategy - 71 -
Chart 8. Time needed to construct a query graph - 73 -
Chart 9. Time needed to compare query graphs - 74 -
Chart 10. Execution time of the original query using the different strategies..... - 74 -
Chart 11. Execution time of each test - 75 -
Chart 12. Time needed to add sources (and update the cache). - 76 -

Table of figures

Fig. 1. SCOUT architecture layers.	- 4 -
Fig. 2. SIM in the Environment Layer[12]	- 6 -
Fig. 3. Android Diagram [4].....	- 7 -
Fig. 4. The progress of the web.	- 8 -
Fig. 5. The Semantic Web Stack[5]	- 9 -
Fig. 6. Example RDF Graph [5]	- 11 -
Fig. 7. Splitting up the query [22]	- 22 -
Fig. 8. Semantic regions.....	- 25 -
Fig. 9. Negative Manhattan distance between semantic regions and a query.....	- 27 -
Fig. 12. Example query graph 1.	- 35 -
Fig. 13. Example of identical query graphs.....	- 36 -
Fig. 10. Notification system of newly encountered sources.	- 42 -
Fig. 11. Simple Query Execution Example	- 43 -
Fig. 14. Query Analysis Class Diagram.	- 45 -
Fig. 15. Query2 Class Diagram.....	- 46 -
Fig. 16. QueryJoinResult Class Diagram.....	- 48 -
Fig. 17. Graph Class Diagram.	- 49 -
Fig. 18. Graph Compare Class Diagram.	- 54 -
Fig. 19. Cache Class Diagram.	- 56 -

List of RDF Data

RDF Data 1: Example of RDF Triple [6]	- 10 -
RDF Data 2. RDF Information in RDF/XML format [5]	- 11 -
RDF Data 3. RDF Information in Turtle format [6].....	- 11 -
RDF Data 4. RDFS Schema Example[7]	- 12 -

List of SPARQL Queries

SPARQL Query 1. Simple SELECT Query. [11]	- 14 -
SPARQL Query 2. Simple CONSTRUCT Query. [11].....	- 14 -
SPARQL Query 3. Simple ASK Query. [11]	- 15 -
SPARQL Query 4. Simple DESCRIBE Query. [11].....	- 15 -
SPARQL Query 5. Example query for analysis.	- 28 -
SPARQL Query 6. Example query analysis: triggers.....	- 29 -
SPARQL Query 7. Example query analysis: query part 1	- 29 -
SPARQL Query 8. Example query analysis: query part 2	- 29 -
SPARQL Query 9. Example query with wrong order.	- 29 -
SPARQL Query 10. Example query using named graphs.	- 30 -
SPARQL Query 11. Example of query 1 after construction.	- 31 -
SPARQL Query 12. Example of query 2 after construction.	- 31 -
SPARQL Query 13. Example of sub query 2 after injection.	- 32 -
SPARQL Query 14. Test query 1a.....	- 60 -
SPARQL Query 15. Test query 1b.	- 61 -
SPARQL Query 16. Test query 2a.....	- 61 -
SPARQL Query 17. Test query 2b.	- 61 -
SPARQL Query 18. Test query 3a.....	- 62 -
SPARQL Query 19. Test query 3b.	- 62 -
SPARQL Query 20. Test query 4a.....	- 62 -
SPARQL Query 21. Test query 4b.	- 63 -
SPARQL Query 22. Test query 5a.....	- 63 -
SPARQL Query 23. Test query 5b.	- 63 -
SPARQL Query 24. Test query 6a.....	- 64 -
SPARQL Query 25. Test query 6b.	- 64 -
SPARQL Query 26. Test query 7a.....	- 64 -
SPARQL Query 27. Test query 7b.	- 64 -
SPARQL Query 28. Test query 8a.....	- 65 -
SPARQL Query 29. Test query 8b.	- 65 -
SPARQL Query 30. Test query 9a.....	- 65 -
SPARQL Query 31. Test query 9b.	- 66 -
SPARQL Query 32. Test query 10a.....	- 66 -
SPARQL Query 33. Test query 10b.	- 66 -
SPARQL Query 34. Test query 11a.....	- 66 -
SPARQL Query 35. Test query 11b.	- 67 -

List of tables

Table 1. Example output SPARQL Query 11. - 32 -
Table 2. Example output SPARQL Query 12. - 32 -
Table 3. Example output SPARQL Query 11 and 12 after joining..... - 32 -
Table 4. Example query result after joining with no shared variables..... - 33 -

Chapter 1. Introduction

This chapter describes the context of this thesis, a description of the problem it attempts to solve, the followed approach, and finally the structure of this thesis.

1.1 Context

Nowadays mobile devices (e.g. smart phone, PDA) have become part of everyday life, not only do they allow people to make phone calls or send text messages; they also allow people to access the Web using a dedicated mobile browser. At the same time, the hardware limitations these devices used to have are fading (e.g. faster processors, more memory and improved connectivity). Although this evolution is making the gap between mobile devices and desktop computers smaller, limitations still exist (e.g. small screen, limited input capabilities), which sometimes hinder the use of the Web on these devices. Next to this, users can not always spend time for looking up information, since these devices are deployed in a mobile environment (e.g. while driving).

It is observed that mobile users are often interested in information related to their surroundings (e.g. find a restaurant nearby where he can eat dish X) and personal preferences they might have (e.g. find a restaurant nearby where he can find his favorite dish). Consequently, if applications are able to automatically take into account the user's environment and personal profile when providing data, the user no longer has to manually look up this information, minimizing the limitations mentioned before (e.g. less time, small screens). Technologies such as RFID and GPS can be employed to make these applications aware of the user's physical environment and the entities in it. When combining this with the improved connectivity of mobile devices, mobile applications are able to access services and information on the Web associated with these nearby objects.

The SCOUT framework, which is currently being developed at the WISE lab of the VUB, supports the development of context-aware mobile applications. These applications are able to offer relevant information and services to the user based on the user's environment and needs at a given time and place. The SCOUT framework achieves this by providing a conceptual and integrated view on the environment called the Environment Model, which is comprised of metadata on the physical entities found nearby the user, together with the user's profile information. The SCOUT framework also allows developers to provide feedback to a user whenever a new source (e.g. a shop) is detected that provides something the user might be interested in (e.g. a shop which sells shoes the user encountered in another store in the past week) by making use of the Notification Service.

1.2 Problem Description

Whenever a new source is detected, the Notification service will try to execute each query, used as a trigger for a certain notification, at both the encountered source, as well as the Environment Model.

A drawback to this approach is that query is executed on the Environment Model, even though the newly encountered source might not contain any information for the given Notification Service Query. Executing queries on the Environment Model, as opposed to executing a query on a single source, is costly. In order to avoid this cost, we first have to find out whether the query actually yields any results on the newly encountered source, before the query should be executed on the environment model.

This thesis also investigates how a query mechanism can be employed to reduce the amount of bandwidth used when the same query is executed multiple times on the Environment Model.

1.3 Approach

In our approach we try to optimize this query mechanism by tackling the issues mentioned in the previous section. For this purpose, we have developed two mechanisms. Firstly, a query pipeline mechanism that is used to create sub queries based on information found in the original query, which can then be executed separately on the newly encountered source, and the Environment Model. Secondly, a cache mechanism which is able to store query results.

First, we extract the relevant sub queries out of the given query. The goal of extracting these sub queries is to allow separate execution of the part of the query which is meant for the newly encountered source only. Only in case the first sub query has retrieved relevant data from the newly encountered source, is the second sub query executed on the Environment Model. Afterwards, these results can be joined to form the actual results for the original query. Because of this extracting these sub queries and constructing them afterwards has its challenges.

Secondly a query caching strategy is employed, which can be used to store query results. More specifically, this query caching mechanism can be used to store the results of sub queries executed on the Environment Model in the query pipeline. Moreover, this system can also be employed for caching query results from Query Service queries. This cache should also take into consideration the smaller memory and more limited performance of mobile devices.

For the query pipeline mechanism, two strategies have been developed, together with a single strategy used for caching. These strategies have been validated and compared through extensive experimentation.

1.4 Thesis Structure

The next chapter provides background information related to this thesis. Topics such as the SCOUT framework, the Semantic Web, Android and caching are discussed.

The third chapter describes related work in the scope of this thesis (i.e. the SCOUT framework, query pipelining and query caching strategies).

The fourth chapter explains the concept of the query pipeline, which is able to process a given query and divide it into sub queries, which can be executed separately.

The fifth chapter explains the Cache that is used to store query results, in relation to the query pipeline explained in chapter five.

The sixth chapter explains the implementation details of all the developed components.

The seventh chapter describes the evaluation of the query pipeline and query caching mechanism, by presenting experiment results and discussing their outcome.

Chapter eight describes the conclusions of this thesis, regarding the use of both the query pipeline and the query caching mechanism, and elaborates on future work.

Chapter 2. Background

This chapter will discuss all the background information required to understand the following chapters. Employed technologies such as the Semantic Web, and Android will be discussed. The first two topics are about the SCOUT framework which is extended in this thesis to support a query pipeline and query caching, and Android, the platform for which this framework is developed.

2.1 SCOUT framework

This section describes the SCOUT framework. First, a small description of the framework is given, together with its main characteristics. Subsequently, an overview of the layered structure of the framework will be discussed, with detailed information about each layer.

2.1.1 Introduction

SCOUT, which is short for Semantic COntext-aware Ubiquitous scouT, is a mobile application development framework which supports the development of context-aware mobile applications. It supports applications which offer relevant information and services based on the mobile user's environment and needs at any given time and place. [1]

The entire framework is scalable, decentralized and distributed so that no centralized access point for data is required and each identifiable entity (e.g. an RFID tag, a location) is responsible for providing and managing its own data and services, by using Web presences which can range from a simple Website/Web service to an online source providing structured information (e.g. RDF files). Due to its open, decentralized and distributed nature (together with its ubiquitous availability), the Web is the ideal platform for deploying these presences. Furthermore, it allows re-use of the wealth of descriptive information already available online (e.g. existing Web pages, RDF information such as FOAF profiles) as Web presences. The SCOUT framework allows a seamless integration and querying of data from several entities, by making use of the Semantic Web standards and vocabularies to describe Web presences (such as RDF data) in a uniform and expressive way. This provides mobile applications with a richer and more complete view on the global environment. [1]

2.1.2 Overview

The framework itself uses a layered architecture, which clearly separates the different design concerns, and to assure the independence between the underlying technologies. Figure 1 gives an overview of the architecture and clearly shows the different layers used: the detection layer, the location management layer, the environment layer and the application layer. [1]

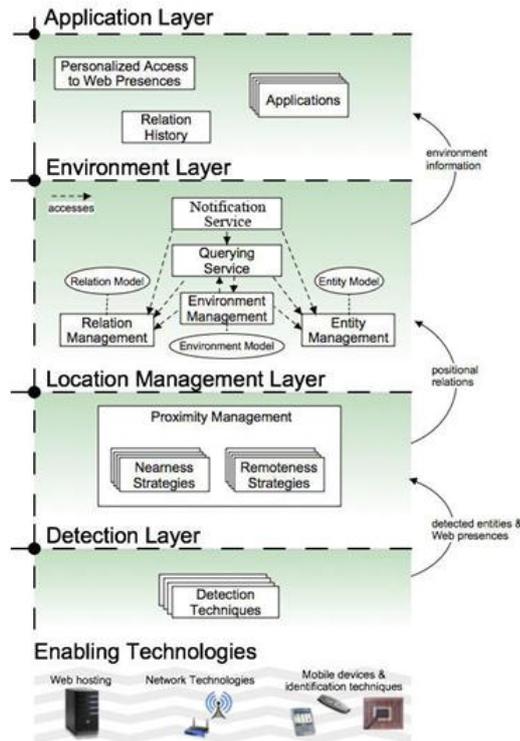


Fig. 1. SCOUT architecture layers.

2.1.3 Detection Layer

The Detection Layer (i.e. the bottom layer) is responsible for detecting identifiable physical entities in the vicinity of the user (e.g. based on coordinates from a gps) and subsequently obtains the reference to the corresponding Web presence. This detection layer contains components that encapsulate different detection techniques (e.g. RFID, NFC, Bluetooth, etc.), which can be used to detect and extract references to Web presences for use by other layers. For example this could occur when an RFID tag is detected in the area and information such as a URI is retrieved from it. This URI contains the location of an RDF file (a Web Presence) containing more information about the object where this RFID tag was located on. Because these detection techniques are encapsulated in a separate layer, and have a uniform interface, the framework can transparently switch from one technique to another (or using several of them in parallel), depending on the available detection techniques and the techniques supported by nearby entities.[1]

2.1.3.1 Location Management Layer

The Location Management Layer is the second layer and is built on top of the Detection Layer. It is responsible for determining which entities are nearby (or no longer nearby) the user. To achieve this it constructs “positional” relationships between the various entities, which are used to express the “nearness” of an entity. In order to determine these relationships it makes use of so-called “nearness” and “remoteness” strategies, which collectively are called proximity strategies. For example, a “nearness” strategy is used when walking past an object with an RFID attached to it. When a short-range RFID reader is used in this example, it is easy to infer the nearness of this RFID tag. [1]

2.1.3.2 Environment Layer

The Environment Layer stores and integrates data about the user and its current environment, and provides services to obtain information from nearby Web presences in both a push- and pull-based manner. The integration of the data is facilitated via the use of Semantic Web technology (RDF/S¹, OWL), while the reasoning capabilities of the Semantic Web standards can be used to infer useful additional data. [1] Semantic Web tools like Jena² and Androjena³ are used to store the data.

Entity and Relational models are used to represent the metadata (or information) of a certain entity (e.g. Web presences, user preferences and characteristics), together with the “positional” relationships the entity has (had) with other environmental entities. Both of these models are maintained and are accessible via a management component, which allows querying and programmatic access to these models.

The core component of the Environment Layer is the Query Service. It allows client applications to query the user’s Entity model, the Relational models and the models of Web presences of (nearby) entities. These queries can range from simple queries retrieving metadata to queries containing complex semantic conditions. [1]

The Notification Service allows applications to obtain references to Web presences of nearby entities in a push-based manner, thus allowing them to become responsive to changes in the user’s environment. An application is able to register itself with this service, allowing it to automatically receive events when nearby entities are encountered or are no longer available. By default, the application is notified of all nearby entities, but the application can specify a filter, after which it will only be notified about the entities that meet the condition specified by the filter (which is encoded as a SPARQL Query). [1]

The last component, the Environment Management component, is used to combine the metadata from multiple Web presences linked via both past and current positional relations, and integrating it with the metadata of the mobile user. This integrated view is called the Environment Model, and it effectively allows the application developer to extensively query any piece of the user’s context and environment. This way the developer is able to query different sources without having to specify them all himself, since the environment manager will determine which sources are relevant and which are not. [1]

An already existing way to optimize the execution of queries is the Source Index Manager, which allows SCOUT to make an index (called the Source Index Model or SIM for short) for any given source and stores metadata information. This information can be used later on to optimize the query process. This optimization occurs because relevant sources for each query can be extracted out of the SIM so that no unneeded sources have to be queried. The SIM is built and maintained in the background during the indexing phase that is triggered whenever Detection Layer detects a physical entity and its online reference has been passed to the Environment Layer. An example of this process can be found in figure 2, if a new source is detected (a.1), it gets send to the Source Index

¹ <http://www.w3.org/RDF/>

² <http://jena.sourceforge.net/>

³ <http://code.google.com/p/androjena/>

Manager. The Source Index Manager then extracts the required metadata and stores it (a.3). Next it will store the data found in the source into a source cache (a.4). [12]

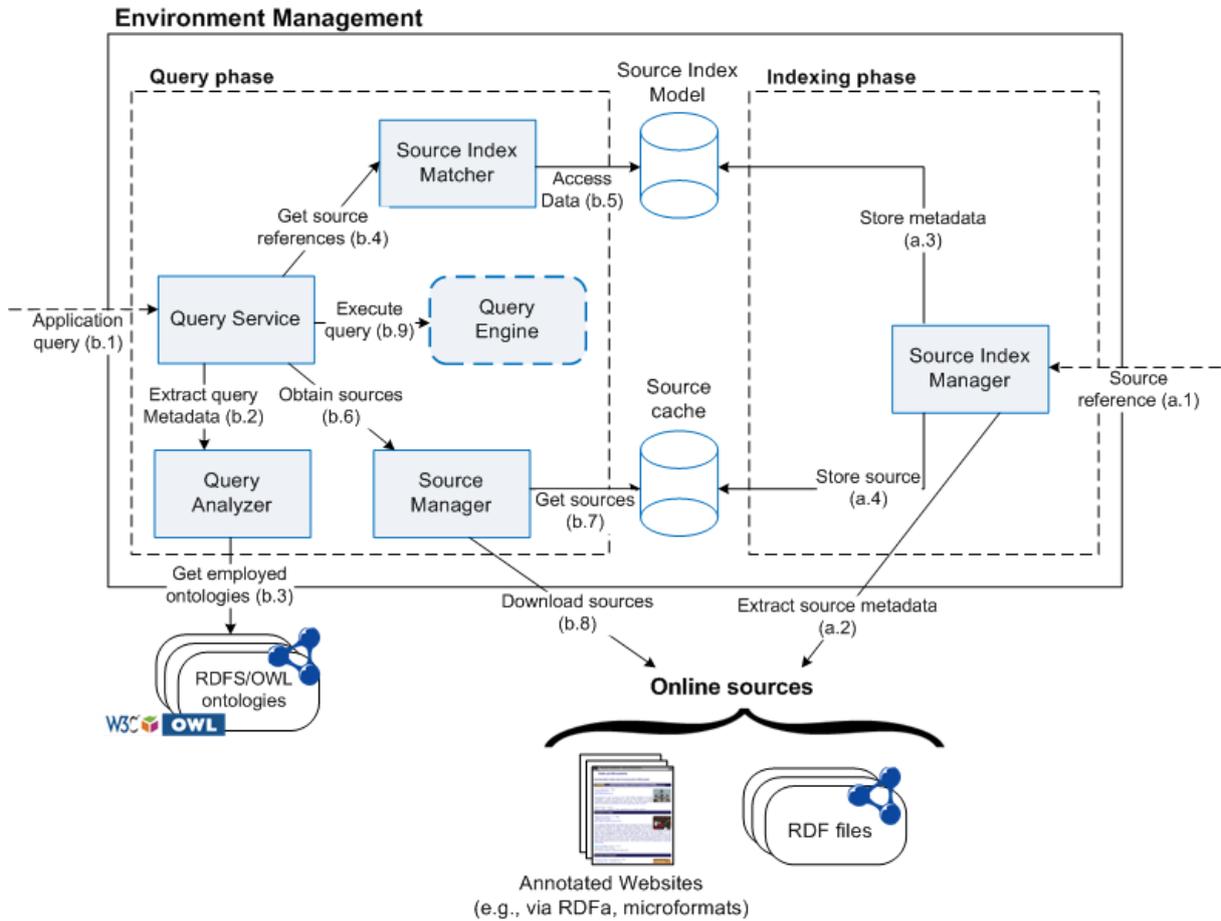


Fig. 2. SIM in the Environment Layer[12]

2.1.3.3 Application Layer

The final layer is the Application Layer, which contains the applications built on top of SCOUT. These applications have access to the services and models provided by the framework. Therefore, they can take both the current and the previous user's environment into consideration when providing services and information to the user. [1]

2.2 Android

Android⁴ is an open-source software stack consisting of different components including a Linux-based operating system, various libraries, the android runtime which adds support for java, an application framework and various applications running on top of it created for mobile phones and other devices.[2][3]

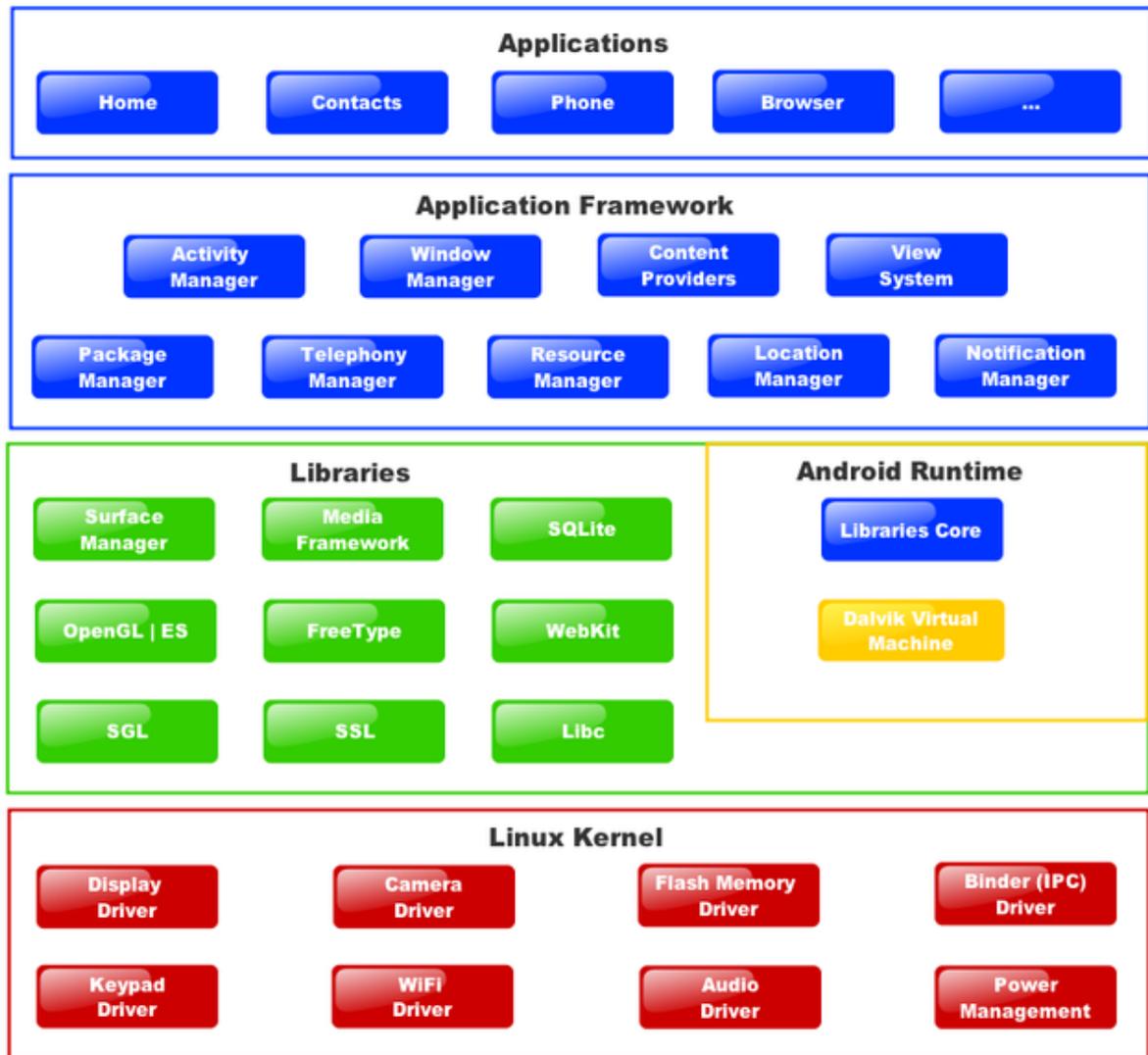


Fig. 3. Android Diagram [4]

Since the SCOUT framework is a Java based framework, which is meant to run on mobile devices and an Android system has all the necessary support for running advanced Java based applications, as opposed to other mobile operating systems which only provide limited support, the Android OS is used for running and testing the SCOUT framework on.

⁴ www.android.com

2.3 Semantic Web

As explained before, the SCOUT framework is a mobile application development framework which supports the development of context-aware mobile applications. These applications offer relevant information and services to the user, based on his environment and needs at any given time and place. [1] To accomplish this, the SCOUT framework makes use of Semantic Web technologies such as RDF and OWL, and Semantic Web tools such as Jena and Androjena. This section explains the rationale and goals of the Semantic Web, as well as a layered conceptual model to show its composition.

2.3.1 The idea and purpose of the Semantic Web

The Semantic Web, which is a crucial component of Web 3.0, is widely considered to be the future of the World Wide Web. As we can see in figure 4, it extends Web 2.0 with explicit content semantics, as Web 2.0 itself extended Web 1.0 with user produced content. More specifically, its purpose is to add semantics (or meaning) to documents, services, and general data on the Web, enabling machines to read and understand these services and information. As a result, machines can use all this information to derive new facts, detect contradictions, and infer new relations and so on. Since search engines are all about understanding the content on the web to improve their search results, they are able to benefit from the Semantic Web to return more relevant results. An example of this is Google Rich Snippets⁵, which makes use of snippets (e.g. RDFa), to know which type of data an element represents (e.g. a location, a restaurant, an address) on a web page. Using these snippets Google can provide more detailed search results, since it knows what each element contains.

As shown in the figure below, the Semantic Web adds extra information such as concepts and relationships between resources to the already existing information found on the Web, thus increasing the expressiveness of the information.

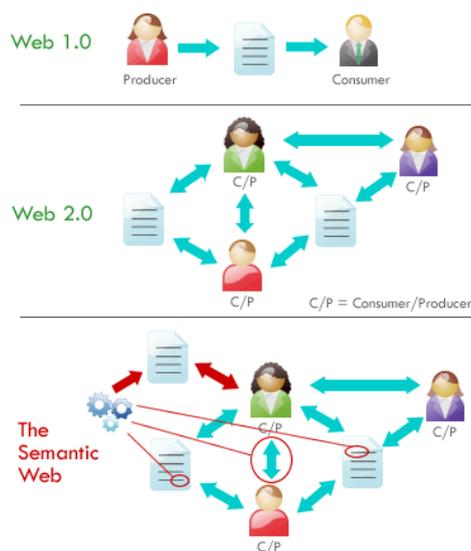


Fig. 4. The progress of the web.

⁵ <http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=99170>

The relationships mentioned before, actually take up the primary role of the semantic web. Because of these relationships we are able to form a Web of Data, instead of just a Web of Documents as the World Wide Web used to be.

2.3.2 The Semantic Web Stack

The Semantic Web Stack, also known as the Semantic Web Cake, describes the layered architecture of the Semantic Web. Figure 4, which can be found below, is a common figure to represent the Semantic Web Stack and its 14 layers.

- The URI/IRI layer uniquely identifies Semantic Web Resources.
- The Unicode layer represents/manipulates text in different languages.
- The XML layer is used as the main interchange structure of data over the Web.
- The Namespaces layer uniquely qualifies the markup from various sources (integration).
- XML Query (XQuery) queries collections of XML data.
- XML Schema helps define the structure (grammar) of specific XML languages.
- The RDF Model & Syntax layer defines RDF triples and represents resource information in a graph structure. It also describes taxonomies based on RDF Schema (RDFS).
- The ontology layer is used to define vocabularies, extending RDFS with more advanced features (e.g. cardinality constraints) and enabling reasoning based on description logic (e.g. OWL).
- The Rules / Query layer describes additional rules via the Rule Interchange Format (RIF) and query RDF (OWL) data based on the SPARQL query language.
- The Logic layer is used for logical reasoning (infer new facts and check consistency).
- The Proof layer explains logical reasoning steps.
- The Trust layer is used for the authentication of sources and the trustworthiness of derived facts.
- The Signature can be used for validating the source of facts by digitally signing the RDF data.
- The Encryption layer can be used to protect RDF data via encryption.
- [5]

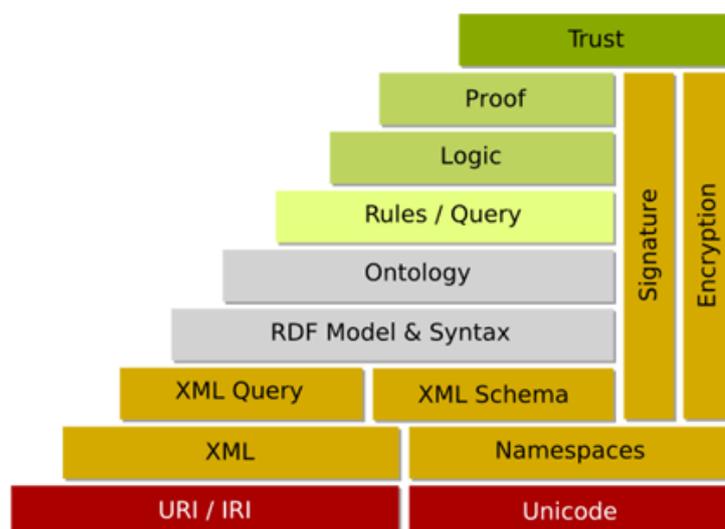


Fig. 5. The Semantic Web Stack[5]

2.4 Resource Description Framework

In the previous sections, RDF (Resource Description Framework) was mentioned as either part of the Semantic Web or because it was employed by SCOUT framework for data representation. This section explains what RDF is and how it can be represented.

2.4.1 What

As mentioned before, RDF is an integral part of the Semantic Web, more specifically the RDF Model & Syntax layer. RDF defines triples of data, representing resource information in a graph structure, RDFS on the other hand describes taxonomies. RDF is responsible for providing a way to describe data models in such a way that the sharing and interchange of data is facilitated.

2.4.2 Representation

RDF is represented by RDF statements (or triples). An RDF statement consists of a subject, a predicate (property) and an object (value) and is represented in the following form “<subject> <predicate> <object>”. Such statements allow us to describe a resource, by providing property values for that resource or other resources to which the given resource is related. Subjects, predicates and objects are all resources, where a Resource itself is anything that can be uniquely identified by a URI (a physical object, a digital content item, etc). There is also a special form of a Resource, which is called a Literal. A Literal is used to represent data such as a String, an Integer ...

The subject of a triple is used to show which Resource we want to describe, the predicate is used for defining a relation between the subject and the object, while the object is the value bound to the subject for that specific relation. In the example below, the subject is “http://www.example.org/index.html”; the predicates are “http://purl.org/dc/elements/1.1/creator”, “http://www.example.org/terms/creation-date” and “http://purl.org/dc/elements/1.1/language”; and the objects are “http://www.example.org/staffed/85740”, which is a Resource, together with “August 16, 1999” and “en”, which are both Literals.

```
<http://www.example.org/index.html> <http://purl.org/dc/elements/1.1/creator>
<http://www.example.org/staffid/85740> .
<http://www.example.org/index.html> <http://www.example.org/terms/creation-date> "August 16, 1999" .
<http://www.example.org/index.html> <http://purl.org/dc/elements/1.1/language> "en" .
```

RDF Data 1: Example of RDF Triple [6]

The data like the example above might not always be very readable. Therefore, this RDF information, which is a directed labeled graph, is also commonly presented and visualized this way, which can be seen in figure 5. The graph shows the collection of triples, with all of the unique URIs and Literals and the relations between them.

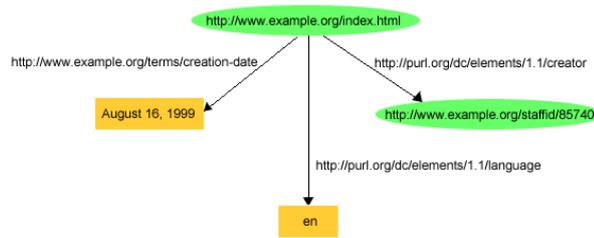


Fig. 6. Example RDF Graph [5]

Various RDF serializations have been created (e.g. RDF/XML, Turtle and RDF N3), because the data might have to be exchanged between several applications and the graph structure, although very powerful, is unsuitable for this exchange.

RDF/XML is the most common representation used for interchanging RDF data between applications, since XML is one of the most commonly used data formats. XML is also one of the most difficult formats to write RDF, compared to others like Turtle and N3 notation. These notations are simpler as they focus are written in a more natural form for the user. The difference between XML and a notation such as Turtle can be seen in the two examples below.

```

<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:extermns="http://www.example.org/terms/"
  xmlns:dc="http://purl.org/dc/elements/1.1/">

  <rdf:Description rdf:about="http://www.example.org/index.html">
    <extermns:creation-date>August 16, 1999</extermns:creation-date>
    <dc:creator rdf:resource="http://www.example.org/staffid/85740"/>
    <dc:language>en</dc:language>
  </rdf:Description>

</rdf:RDF>
  
```

RDF Data 2. RDF Information in RDF/XML format [5]

```

<http://www.example.org/index.html> <http://purl.org/dc/elements/1.1/creator>
<http://www.example.org/staffid/85740> .
<http://www.example.org/index.html> <http://www.example.org/terms/creation-date> "August 16, 1999" .
<http://www.example.org/index.html> <http://purl.org/dc/elements/1.1/language> "en" .
  
```

RDF Data 3. RDF Information in Turtle format [6]

2.5 RDF Schema

As mentioned before, RDF Schema is used to define grammars as opposed to RDF which describes resources and the relations between those resources. RDFS is a vocabulary description language, allowing developers to create vocabularies defining properties (or relations) and types (or classes), which can be re-used in RDF statements. Furthermore, in RDFS subclass relations between classes can be defined, as well as subproperty relations between properties, and the allowed types of subjects (domains) and object (ranges) of a certain property can be specified.

Next to that it is used to provide the basic elements for the definitions of ontologies, which are described in the next section. The example below shows the definition of two classes (“Resource” and “Class”) together with one property (“type”) and is used to define a small part of the official RDF/RDFS definition.

```

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#">

  <owl:Ontology rdf:about="http://www.w3.org/2000/01/rdf-schema#" />

  <rdfs:Class rdf:about="http://www.w3.org/2000/01/rdf-schema#Resource">
    <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#" />
    <rdfs:label>Resource</rdfs:label>
    <rdfs:comment>The class resource, everything.</rdfs:comment>
  </rdfs:Class>

  <rdf:Property rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#type">
    <rdfs:isDefinedBy rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#" />
    <rdfs:label>type</rdfs:label>
    <rdfs:comment>The subject is an instance of a class.</rdfs:comment>
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class" />
    <rdfs:domain rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource" />
  </rdf:Property>

  <rdfs:Class rdf:about="http://www.w3.org/2000/01/rdf-schema#Class">
    <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#" />
    <rdfs:label>Class</rdfs:label>
    <rdfs:comment>The class of classes.</rdfs:comment>
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource" />
  </rdfs:Class>

</rdf:RDF>

```

RDF Data 4. RDFS Schema Example[7]

2.6 Web Ontology Language

The Web Ontology Language, also known as OWL, is designed for use by applications that need to process the content of information, and not for applications which just present information to humans. OWL facilitates a greater machine interpretability of Web content, then supported by only XML, RDF, and RDF Schema, as OWL facilitates provide an additional vocabulary along with a formal semantics. To accomplish all of this, OWL provides three increasingly-expressive sublanguages: OWL Lite, OWL DL and OWL Full. [8]

OWL Lite is the weakest of the 3 sublanguages, providing a classification hierarchy and some simple constraints.

OWL DL provides a maximum of expressiveness, while still retaining computational decidability.

OWL Full also provides a maximum of expressiveness, but compared to OWL DL it provides no computational guarantee.

A large number of existing and well-known ontologies already exist, allowing any RDF data provider to extensively describe resources in a dataset. Some commonly used ones are Friend-Of-A-Friend⁶, the Dublin Core Metadata Initiative⁷ and GeorSS⁸ ...

⁶ <http://www.foaf-project.org>

⁷ <http://dublincore.org>

⁸ http://www.georss.org/Main_Page

2.7 SPARQL

SPARQL⁹ is a query language for RDF and is used extensively in this thesis. This section will explain what it is, where it is used and what its syntax is.

2.7.1 Introduction

SPARQL, which is a recursive acronym that stands for SPARQL Protocol and RDF Query Language, is an RDF query language. Although other query languages exist, SPARQL is standardized by the RDF Data Access Working Group, which is part of the World Wide Web Consortium and is considered a key Semantic Web technology. Currently it is also the only query language supported by the SCOUT framework. [9]

The SPARQL standard also specifies how SPARQL clients can communicate with a SPARQL endpoint/processing service (i.e. using WSDL 2.0¹⁰), together with how the query results should be returned to the entity that requested them. [10]

2.7.2 Syntax

The SPARQL language specifies four different query forms (SELECT, CONSTRUCT, ASK and DESCRIBE). Although the query language has similarities with SQL, it doesn't support all of its query forms such as CREATE, DELETE, INSERT, nor does it allow sub queries and aggregate functions. It also works different internally since it is a graph pattern matching language built on top of RDF.

The SELECT query form is used to request RDF terms (blank nodes, URI's, or Literals) from an endpoint and bind them to variables, which are directly returned as simple bindings, that represent pattern matches, just as they would in an SQL select. These bindings can be represented as a table, where each column is a requested variable, and each row contains the bindings (values that matched the entered pattern matches) for the selected variables. SPARQL Query 1 shows a select query which will return the people who know each other, together with the nick of the second person if available, which is represented by the OPTIONAL clause.

RDF Data:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

```
_:a foaf:name "Alice" .
```

```
_:a foaf:knows _:b .
```

```
_:a foaf:knows _:c .
```

```
_:b foaf:name "Bob" .
```

```
_:c foaf:name "Clare" .
```

```
_:c foaf:nick "CT" .
```

SPARQL Query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

```
SELECT ?nameX ?nameY ?nickY
```

```
WHERE
```

```
{ ?x foaf:knows ?y ;
```

```
  foaf:name ?nameX .
```

```
  ?y foaf:name ?nameY .
```

⁹ <http://www.w3.org/TR/rdf-sparql-query/>

¹⁰ <http://www.w3.org/TR/wsdl20/>

```
OPTIONAL { ?y foaf:nick ?nickY }
}
```

Result:

nameX	nameY	nickY
"Alice"	"Bob"	
"Alice"	"Clare"	"CT"

SPARQL Query 1. Simple SELECT Query. [11]

The CONSTRUCT query form is used to create a single RDF graph specified by a graph template. Its result is an RDF Graph formed by substituting each variable in the graph templates using the results from the given query (which is similar to the SELECT query mentioned before), and combining the newly formed triples by a set union into a single RDF graph. This is both a powerful and easy mechanism which allows new data sources to be created from existing data. SPARQL Query 2 shows a construct query which creates vCard¹¹ properties from existing FOAF information. [11]

RDF Data:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

```
_:a foaf:name "Alice" .
```

```
_:a foaf:mbox <mailto:alice@example.org> .
```

SPARQL Query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

```
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
```

```
CONSTRUCT { <http://example.org/person#Alice> vcard:FN ?name }
```

```
WHERE { ?x foaf:name ?name }
```

Result:

```
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
```

```
<http://example.org/person#Alice> vcard:FN "Alice" .
```

SPARQL Query 2. Simple CONSTRUCT Query. [11]

The ASK query form can be used by applications to test if a certain query pattern has a solution. Apart from whether or not solutions exist, no other information is returned about the possible query solutions. SPARQL Query 3 is an ASK query which will check if a "person" with name "Alice" exists.

RDF Data:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

```
_:a foaf:name "Alice" .
```

```
_:a foaf:homepage <http://work.example.org/alice/> .
```

```
_:b foaf:name "Bob" .
```

```
_:b foaf:mbox <mailto:bob@work.example> .
```

¹¹ <http://en.wikipedia.org/wiki/VCard>

SPARQL Query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
ASK { ?x foaf:name "Alice" }
```

Result:

Yes

SPARQL Query 3. Simple ASK Query. [11]

The DESCRIBE query form is used to create a single result RDF graph containing RDF data about resources. The data returned is not prescribed by the given SPARQL query, but by the SPARQL query processor, which was used to execute the query with. The DESCRIBE query form takes each of the resources identified in a solution, together with any resources directly named by a URI, and creates a single RDF graph by taking a “description” which can come from any information available, including the target RDF Dataset. [11]

SPARQL Query:

```
PREFIX ent: <http://org.example.com/employees#>
DESCRIBE ?x WHERE { ?x ent:employeeid "1234" }
```

Result:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0> .
@prefix exOrg: <http://org.example.com/employees#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#>
```

```
_:a exOrg:employeeid "1234" ;
```

```
    foaf:mbox_sha1sum "ABCD1234" ;
    vcard:N
      [ vcard:Family "Smith" ;
        vcard:Given "John" ] .
```

```
foaf:mbox_sha1sum rdf:type owl:InverseFunctionalProperty .
```

SPARQL Query 4. Simple DESCRIBE Query. [11]

To add some more expressiveness to the query, there are a couple of modifiers which can be used in a query: REDUCED, DISTINCT, ORDER BY, OFFSET, LIMIT, FILTER, OPTIONAL, and UNION. Since these are less important only a couple will be explained and for the others you can go to reference [11].

DISTINCT will make sure no duplicates are available in the result from the query, ORDER BY will make sure the results are ordered, based on the order selection provided. A FILTER can be used to reduce the amount of results returned by a query, by adding one or more conditions (restrictions) which have to be met. For example, this could be a request for information about people in a golfclub where only people older than 50 years old should be returned.

2.8 Semantic Web Frameworks

Semantic Web Frameworks have been created to provide abstractions for using parts of the Semantic Web in different programming languages. These frameworks provide a programmatic environment for use with various Semantic Web technologies, including RDF, RDFS, OWL, SPARQL and more. One example of such a framework is Jena¹², which is a Semantic Web Framework for Java which contains the following:

- A RDF API
- Reading and writing RDF in RDF/XML, N3 and N-Triples format
- An OWL API
- In-Memory and persistent storage (e.g. RDF)
- SPARQL Query Engine

Since the SCOUT Framework is meant to be used on Android devices, which don't support all Java language features (and also have resource restrictions), the Androjena¹³ framework will be used which is a port of Jena for use on Android.

2.9 Caching

Since caching is a major aspect of this thesis, this section will give a short introduction, followed by why caching is needed and what kind of data we can cache.

2.9.1 Introduction

Caching is a mechanism for using a temporary storage of data to, for example, reduce bandwidth usage, server load, ... In the case of mobile devices, this can be used to reduce costs and time for the end user to find certain data, especially when the same data is requested multiple times. Apart from speeding up the search process or reducing costs, such as internet costs, there are also some possible downsides to caching. For example, the data might not always be up to date, or if you never look up the same kind of data, the cache would actually be useless and only take up space on the device or its memory. To prevent this from happening two important properties are used: temporal locality and spatial locality, which both exploit the way data access works.

Temporal locality is the property that items that have been referenced recently are likely to be referenced again in the near future. Examples of this are FIFO (First In, First Out), which will remove the first element that got added to the cache, if space has to be made for newer elements, LRU (Least Recently Used) where the least recently used cache units are considered to be the least useful ones, and LFU (Least Frequently Used) in which the least frequently used units are considered to be the least useful ones.

Spatial locality is the property that if an item has been referenced, other items that are physically close to it are also likely to be referenced.

2.9.2 For what data

Caching can be used to store a lot of data types. In the case of the SCOUT framework it can be used to store entire sources in RDF format, or just tuples retrieved from a source. This thesis handles the

¹² <http://jena.sourceforge.net>

¹³ <http://code.google.com/p/androjena>

caching of query results, which is an extension to the already available caching mechanisms available in the SCOUT framework.

2.9.3 How does it work

To show how caching works, a simple example using a web cache (e.g. a browser cache) will be used.

A web cache is used to reduce the amount of requests done to one or more servers (web pages, css, javascript,...). To achieve this goal, it stores the different types of data locally in its cache (after retrieval from the server), so it can later on retrieve the data faster. However some complications occur in a web cache: e.g. limited cache space, dynamic web pages which have to be updated after a specific time, ... Therefore, various caching mechanisms have been created to take into account such issues. Some of these issues, together with solutions can be found in the next chapter.

In its simplest form, a cache works as follows: after a request is done, the cache will first check whether it contains the required data. To achieve this it makes use of the following two commonly used terms: cache hit and cache miss. A cache hit means that the required data is found in the cache (and this data has not 'expired' yet). A cache miss means the data couldn't be found (or the data has 'expired'), so the request has to be sent to the server in order to obtain the required data. Afterwards the data is added to the cache or the already existing (expired) data in the cache is updated to the latest version. Since cache space might be limited, cache eviction strategies are employed to remove the least useful data from the cache. Some examples of these strategies are LRU, which will remove the data from the cache that hasn't been accessed in the longest time, and LFU (Least Frequently Used), which will remove the data from the cache that was used least frequently.

2.9.4 Query caching

Since the goal of this thesis is to optimize the query execution by adding query caching, this subsection explains what query caching is.

Compared to the previous subsection, query caching doesn't store entire files (or sources), but instead stores the results of each query, together with an index (i.e. the query associated with these results) to check if the cached results might be a match to a (newly) given query. Using this mechanism, only relevant data is stored in the cache, where file caching would always store all the data, instead of only the data required.

Chapter 3. Related Work

This chapter describes existing work related to the topics discussed in this thesis. Related work is split up into different parts according to the handled topic, covering the SCOUT framework, query pipelining, and caching mechanisms.

3.1 The SCOUT framework

As mentioned before, the SCOUT framework exists out of many parts (layers), each with their own purpose. In the SCOUT framework the user's environment consists of a number of physical entities, each with associated Web Presences, which can then be used in queries to provide personalized information. Each Web Presence stores information or services related to a physical entity (for instance, a Website/service, (online) RDF data, ...). All this info can then be used so queries can reference the metadata of any previously encountered entity. The metadata of all physical entities mentioned before are then combined with the user's profile. This user profile, which got build over time, together with the metadata of all these physical entities is used to create the Environment Model which applications can then use. The following subsections will go into some detail regarding to related work of some of these mentioned parts.

3.1.1 Linking of physical entities to web presences

Several projects exists that focus on the linking of physical entities to digital information (such as Web presences mentioned before). One of the first projects to focus on using this kind of linking is the HP Cooltown project [1, 12]. The goal of the Cooltown project was to provide an infrastructure for nomadic computing, in which "nomadic" refers to humans moving between places (e.g. home, work) and "computing" refers to the services provided to these users. In particular, services integrated with entities in the user's environment. The Cooltown project has two objectives to achieve: 1) each entity in the user's environment needs to be associated to a web resource (also called a Web Presence), these resources could go from simple web pages to web services. 2) Achieving a high degree of interoperability for interactions with devices. For this second objective the eSquirt protocol was developed to provide a high interoperability.[39] This project is similar to the SCOUT framework as it tries to connect physical entities in the user's environment to a Web Presence. However, the Cooltown project only allows you to retrieve the information for this specific physical entity, while the SCOUT framework not only allows you to retrieve information but also to query for data related to this entity (if needed).

An example of a commercial application using this type of technology is Touchatag¹⁴ from Alcatel-Lucent Ventures, which makes use of RFID technology to connect objects to online applications. An example of this is the Web Link Wheel application, in which you can store a series of URLs. After you have stored them you can use an RFID tag to go through these links. Opening them one by one, till you reach the end of the list, after which you can start again from the top of the list. In [13], an open lookup framework is discussed which uses RFID tags to link objects to resource descriptions, thus allowing users to retrieve information on specific tagged objects. The basic idea behind this approach is the connection of pieces of digital information, to a specific latitude-longitude coordinate via a mobile device. For example, an implementation of this is the GeoNote¹⁵ system which works in the following way. First, a user has to enter some information for the note he wants to create: a title, a

¹⁴ <http://www.touchatag.com>

¹⁵ http://www.ercim.eu/publication/Ercim_News/enw47/persson.html

recipient [which is not required, but can be single individual or a group of people], a signature [used for identification of the user] and a placement label in case of bad geographic location data (such as faulty gps coordinates) or just to provide additional information. A person can access other people's notes "placed" nearby his personal location, filtering the available notes via personal preferences. For this reason a pull and push based system is used. Another way to retrieve information is by using notifications, which also make use of the pull and push based system mentioned before. These notifications can be based on certain filter settings, which will alert the user if something is present where he might be interested in, based on his personal preferences.

While most of the projects mentioned before focus on just retrieving data related to a physical entity or only use one specific detection method (such as RFID or GPS), the SCOUT framework tries to find a balance between all of it. The Detection layer is in charge of detecting the connections between a physical entity and a Web presence, using a single detection method or using multiple detection methods together. With the Environment layer the SCOUT framework integrates all data, so that queries are able to refer to both the information found in the user model as well as the surrounding entities. The Environment layer is also able to forward notifications when certain conditions are met but those conditions are not limited to the notification system in the GeoNote project.

3.1.2 Location specific data storage

As mentioned in [1] many existing approaches focus on the location-specific retrieval of information using a centralized Information System (IS). In these approaches an IS is used to store and maintain all location-specific information. Some examples of this are mentioned in [14-16].

In [16] shopping is used as an example. In this example they want to know more about a certain product, in that case it would work as follows: first the product identification from a certain product is read, either via RFID or barcode scanner. Based on this product identification, a query is fired at a Search Service (IS) which then queries the various resource repositories (IS, that is filled by vendors and web crawler which tries to identify new sources that might be useful) for information about that product and then returns the results back to the client.

Although this system seems to be good for simple data retrieval, it doesn't provide any other semantic information apart from the information specific to e.g. a single product. In our aforementioned example, it is not able to search for related products or able to make connections to products looked up before. This makes it clear that although the data search process is specific it isn't context aware. The SCOUT framework tries to accomplish this, not by providing only information about a specific physical entity, but also by providing information about semantic relations for e.g. that product if required. In the aforementioned example such a semantic relation could be the retrieval of one or more related products based on the type of product.

Besides providing information about semantic relations, the SCOUT framework also use a decentralized model, which re-uses existing online resources, that makes it easier for each client to look up information. Therefore content providers are able to do with their data what they want, thus increasing control over their own data and making it easier for more applications or services to use their data. Because the SCOUT framework runs on the client there is also no single point-of-failure (while the Search Service in the previously mentioned example is).

3.1.3 Integrated view of context sources

In contrast to the previous subsection, where location specific information is stored in one place, other approaches exist that provide a centralized (or integrated) view over a set of distributed data sources.

Such context-aware middleware approaches often rely on context providers for retrieving information about the user's environment [17-19]. These providers use both internal (e.g. sensors) and external (e.g. a traffic service) sources to extract the required data, which then can be accessed using a context-aware application. The difference with SCOUT is that these approaches usually are not using existing online data sources, but use other approaches, such as directory-like services that applications can use to find relevant sources to query themselves, or by providing a single unified view on the user's environment, which is done by integrating various context sources. [17-19].

An example of the integration of context sources is [29], which describes "nRQL". "nRQL" uses a Knowledge Base, much like the Environment Model in SCOUT, to provide answers to a given query. A major difference to SCOUT is that "nRQL" only makes partial use of certain Semantic Web Technologies, where SCOUT is trying to have full support for. Because of this "nRQL" uses its own type of "concrete" states instead, representing individuals and their relationships (called 'ABoxes'), which is what RDF and OWL provide. OWL is supported by "nRQL" by converting the (required) data to an 'ABox'. Another major difference is that "nRQL" runs in a server environment, while the SCOUT framework runs on mobile environments.

Another example of the integration of context sources is Haystack, an open-source RDF-based information management environment [20]. It combines data found in RDF sources with metadata created locally by the user, and represents this data to the user in a visual way. Next to this, metadata from previously encountered sources is also stored, to create links between the data in these sources and current sources.

Just as mentioned in the previous subsection, where a query is fired at an Integrated Search service, which then returns the results, most of these approaches make use of an external service, that handles the queries fired at certain data sources. The advantage of this is that the retrieval and processing of the data is offloaded to the external service, thus providing an efficient and high-level access to (dynamic) data. However, a drawback of such services, is that they are not very flexible or scalable, since every query has to be handled by this service. Also these services are only able to return results from sources explicitly pre-registered to them. Another downside is that these services can only access other services which implement a supported interface to access their data, so it is not possible to query single RDF files, which is something the SCOUT framework is able to do.

Since mobile devices are getting more powerful (for instance, regarding processing power and memory capacity), the need to use these services is reduced, allowing the way for solutions, such as the SCOUT framework, to provide a client side alternative to these otherwise server side services. For this purpose, SCOUT constructs and maintains an integrated view, called the Environment Model, on the mobile device itself. As mentioned before this makes it so that no single point-of-failure exists, and also allows the SCOUT framework to be more flexible in what sources can be used.

3.1.4 Determining relevant sources for a given query

Various approaches exist which keep source indices in order to optimize the query retrieval process. Most of these approaches focus on optimizing query distribution across query endpoints [23] or keeping intensive instance information found on the data [24, 25].

A first example of such an indexing mechanism can be found in [21]. Which uses very specific data for its index, such as a location (e.g. gps coordinates), to retrieve data from a Voronoi Diagram-based index. This index records information about the closest regions corresponding to a set of geometric point and thus only focuses on a single feature when indexing.

Another example is [22] that, given a query, tries to identify an appropriate set of information sources and afterwards formulates and executes the appropriate sub queries based on which data can be found in a certain index. To achieve this it looks up whether certain classes (requested in the query) can be found in an information source. If so, they are included in the query execution.

This last part is what the SCOUT framework tries to achieve, by storing metadata about encountered sources in a balanced way between a minimal overhead and still guaranteeing a high selectivity. The major difference is that the SCOUT framework provides multiples strategies for storing metadata, as opposed to indices as mentioned in the previous examples. This metadata can contain information about only classes or only relations while others use a combination of both. As a result, the indexing in the SCOUT framework is flexible, compared to some of the work mentioned before.

3.2 Query Pipelining

To improve performance when querying multiple data sources, or retrieving different types of data there exists multiple pipeline approaches such as mentioned in [21, 22]. These approaches execute the retrieval process in multiple steps (or processing units). Such processing units can either be executed one after the other, for example when the second unit needs to use the results from the first unit, or executed simultaneously when no results between units are required.

In [21] an example of method 1 is mentioned. Whenever a client needs information, it submits a query to a server and then waits for an answer. If the client arrives in a new cell before the result is retrieved, which means the data it requested before it becomes invalid, a new query is executed after the results have been returned from the server. This process is then repeated till a valid query result is returned. In this example, each new query execution can be seen as a processing unit, which keeps getting repeated until the answer is found.

An example of method 2 is [22] which, based on the given query, will detect the classes of the data requested in the query. For each class (or type) it looks up the sources where data of this type can be found. An example of this can be seen in figure 7, where the query has been split up into the various classes (e.g. Port, Ship, Seaport) for which sources need to be looked up for. Then it executes the queries for each source in parallel. Once all queries are executed the results are joined, providing a result set.

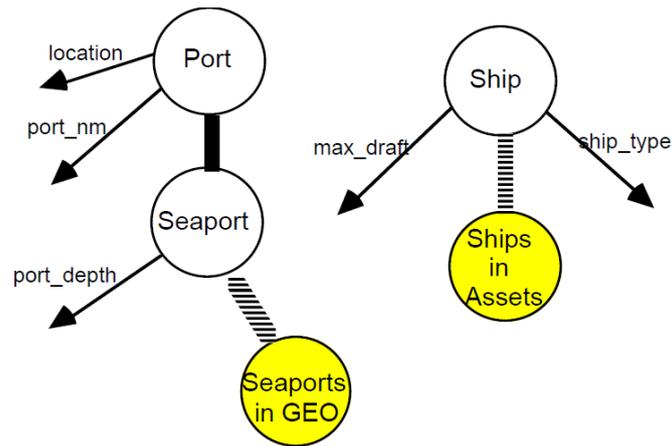


Fig. 7. Splitting up the query [22]

The focus in this thesis will be on method 1, where a pipeline is created to execute different parts of the query one after the other based on their results. In the SCOUT framework, so-called notification queries mostly exist out of two parts. The first part always has to be executed (e.g. retrieve information about my current location), while the second part (e.g. retrieve information about products I encountered in previous locations if the current location is a store), should only be executed if the previous part had any results. This way, when the first part returns no results, unnecessary processing power can be spared, since the second part which has to be executed on the environment model (e.g. multiple sources), is more costly than the first part. Moreover, only the second part can be cached, which is based on already visited sources, since the first part always has to be executed on the newly discovered source.

3.3 Caching mechanisms

Since part of this thesis investigates how caching can be improved within the SCOUT framework. More specifically how (sub-) query results can be cached for re-use in other queries.

3.3.1 Client side caching mechanisms

As partially mentioned in the previous subsection there are various caching mechanisms available already for client side caching. In [20] and [36] some of these caching mechanisms are mentioned with the focus on explaining how Semantic caching works and how this is a possible improvement over the already available mechanisms.

A first (and very basic) example of a caching mechanism is used in [20] for caching, where in the case of read-only RDF files, the entire source file is downloaded and stored locally in the cache for later reuse by the browser.

Another well-known mechanism is page caching [36], which is used for storing entire pages full of tuples. These are the simplest to manage since they have a fixed size and are accessed in a nearly identical way as a traditional page-based database buffer. For an index is kept, to look up results this index has to be used to check if any pages contain the results the query is looking for. If no such results are found, then a request has to be sent to the server for the required pages. The downside to this mechanism is that there is a specific overhead of data available in the cache, so that less operational memory is available for other work.

Another well-known mechanism is tuple caching [36], where individual tuples are cached instead of pages of grouped tuples. This allows maximum flexibility in the tuning of cache contents to access locality properties of applications. The downside to this mechanism is an increased overhead, since for every missing tuple a request has to be sent to the server to retrieve the missing information. A solution for this is also mentioned [36], which first looks for all the missing tuples and then forms a single request to the server for all missing tuples, instead of requesting tuple by tuple. Another overhead issue that might arise is the fact that extra information has to be stored to know what each tuple contains, compared to the mechanism mentioned in the previous paragraph.

The caching mechanism used in this thesis is Semantic caching (also called query caching) which, just as page caching, provides a means for a cache manager to aggregate information about multiple tuples. It does this by building up semantic regions, which, compared to page caching, can change dynamically in size and shape. To retrieve data this caching mechanism makes use of two types of queries. The first one is a probe query, which retrieves information in the cache related to the main query, while the second query, the remainder query, will retrieve any missing information from the server. Examples of this type of caching are given in the following subsection.

3.3.2 Caching query results

As mentioned in the previous subsection and [36], query caching makes use of two types of queries, one being a probe query, which is able to retrieve data from the cache related to a given query, and the second being a remainder query, which, based on the results returned using the probe query, will create a new query which retrieves the results that were not found in the cache for the given query. The following paragraphs will explain various forms of query caching as discussed in [21, 22, 26-32, 36-38].

The proposed mechanism in [21] uses a “simple” index based on the region (e.g. the location) the current user is in. The region serves as a base to check if there is relevant information in the cache for a given query. This is a simple process, since the queries only have to retrieve a limited amount of data, based on the current region the user is in, which can easily be indexed to retrieve the required information out of the cache.

In [22] query results are stored per source, taking into account information about the classes available in the query results. For this it stores meta-information, about these classes, so that it can easily detect which information is already available in the cache, and which needs to be requested from other sources. Although this approach stores query results, the results of the full query are never stored. Query results are only stored for the sub queries, which are generated based on classes found in the main query. This makes it less complicated for knowing what is available in each cached query result, because it does not have to take into account relationships between two classes and the extracting of any data it might need out of the cache.

The mechanism proposed in [26] takes into account query patterns when looking up information in the cache, while only storing the compound cache objects, related to these query patterns, in the cache. Afterwards, based on the data provided in a query pattern, these compound objects can be extracted and used wherever it is required. The cache makes use of an MD5 hash of each query for easy comparison and retrieval of cached results, but is restricted to simple RDF and doesn't take into account any RDFS information (e.g. relations, subclasses, properties, inverse relations) during the comparison of queries. Therefore, only identical queries can be compared and thus the comparison

doesn't take into account queries which are semantically the same (i.e. contain the same constraints) but might have been structured differently (e.g. different variable names, or different order of triple patterns). As opposed to the mechanism in this thesis, where such syntactical variants will be taken into account.

The proposed caching mechanism in [27] caches intermediate results of the query execution process. This way, similar query parts that are re-used across queries are stored in the cache, allowing for more fine-grained caching. It relies on using AET's (short for Algebra Expression Tree) to achieve normalization, allowing graphs to be compared in a less strict way than mentioned above. More specifically, in case graphs are semantically equivalent (i.e. contain the same data constraints) but have a different variable naming schema, AET normalization allows them to still be recognized as being equivalent. AET normalization converts logically equal AET's (obtained from a given query) to a uniform AET. It does this by performing a pre-order traversal in which variables are labeled with sequential integers. After this step, it is able to identify equivalent query execution plans with unique query logic, without having to take into account specific variable naming.

In [28] a caching mechanism is proposed which can be used for DHT-based (Distributed Hash Table) RDF databases, which are used to store RDF triples in a distributed manner. The main focus of such databases is to distribute the storage as well as the query load over several nodes in a peer-to-peer network to improve scalability. Their caching mechanism is comparable to the approach mentioned in the previous paragraph, taking into account syntactic variants of semantically equivalent queries. Each query is first converted to a graph structure, and specific variables names are converted such that they can easily be compared. Next to comparing the structure of a graph, intermediate results are hashed, which can later on be used to further improve the performance. This caching mechanism also performs a bottom up search for intermediate results, to take into account subquery results of previously executed queries so that, just as in the previous paragraph, (partial) results can be reused instead of having to execute the entire query again. As a consequence, only data that was missing has to be requested.

In [29] the mentioned "nRQL" engine can be advised to maintain a so-called query repository, which caches the results of previously executed queries. This query allows cached results to be reused if the same query is executed again.

In [31] a semantic caching approach is described which takes into account the relations of semantic containment as well as intersections between a query and its corresponding cache items.

Both [30] and [36] describe Semantic caching, which is a synonym for query caching, by defining semantic regions in the cache and defining relations between these regions. Each region contains a constraint formula (which describes the available data), a set of tuples, a pointer to the list containing these tuples and some extra information for the eviction strategy (e.g. a 'score'). The difference with the caching approaches mentioned before is that these semantic regions can intersect so that less space is used because tuples aren't stored more than once. Therefore, more data can be cached in the same amount of space as one would be able to use the other mechanisms. An example of this can be found in figure 8 in which can be seen that there is only 1 semantic region Q1 after the first query. After the second query Q1 intersects with Q2 because of shared tuples and the results in Q1 become fragmented, to avoid duplication of tuples between Q1 and Q2, and after

the third another intersect gets added between Q1 and Q3, which leads to more fragmentation of results.

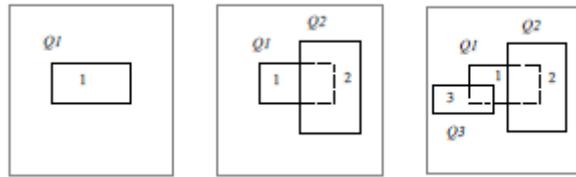


Fig. 8. Semantic regions

In [32] various methods for query matching are described which can be used in Semantic caching to determine which parts of which region are useful. This is done by storing semantic views, which contain metadata, for each Semantic region which can then be used to compare with a new query. The first method is the simplest one, only taking into account an exact match between an input query and the semantic view. The second method is used when no exact match can be found. In this case, it looks for a match as close to the given query as possible (e.g. a view also containing some extra answers). For this purpose, it makes use of various algorithms, such as “BestContainingMatch”, which looks for a containing match from which extra answers can be filtered with minimal effort, and “BestOverlappingMatch” which will look for a match that overlaps with most of the given query using a simple heuristic. A third method is called knowledge-based matching, which, based on semantic knowledge, tries to infer related data from existing data. And the fourth and last method, the BestMatch algorithm, will try to combine all the previous methods to form its result.

In this thesis, the approach of query caching similar to the one discussed in [27] will be used. For each given query a graph is constructed, corresponding to the graph pattern described in the query. The major difference with [27] and most of the others mentioned before, is that not only existing relations will be taken into account; extra information such as inverse relationships or subclasses and sub properties (as defined in RDFS) is considered as well.

3.3.3 Cache invalidation techniques

Invalidation techniques are used to determine when certain data in a specific cache is deemed to be invalid, and thus can't be used anymore in specific situations.

In [34], various cache invalidation schemes are discussed which can be used for location dependent updates. The idea behind it is to remove data from the cache that might not be needed anymore, since the user is at a different location where this data has become useless. This makes it so that the cache doesn't have to worry about redundant information and thus can increase performance and reduce memory usage.

The first method explained is BVC (Bit Vector with Compression). In this method the complete validity information is attached to a data item value. This means that the complete set of cells (locations) in which this data is valid is stored in the cache. The BV (bit vector) for a single item has the same length as the amount of cells available in the cache. When the value for a single cell is “1” this means that the data item is valid for that specific cell. The validity checking algorithm for this works as follows: whenever the value of a data item is required for a location-dependent validation, the client will listen to the broadcast for the current cells, which use an ID and CID (cell ID). If it finds that the data is valid for the CIDth bit then the data can be used, otherwise the data is deemed

invalid. The downside to this method is that it can cause a lot of overhead when a lot of data is present, since for each data element a new BV has to be created, which is as long as the amount of cells available even if that data item would only be valid for a single cell.

The second method is called GBVC (Grouped Bit Vector with Compression), which is an extended version of the first method to reduce the overhead. It does this by only keeping track of some of the validity information for some of the adjacent cells. The reason for this is that a mobile client in most cases will remain in the same cell (location) or the cells adjacent to it. And even if the client moves away it will take some time, so that the data probably has been invalidated automatically already by the server. Compared to the previous scheme, the GBVC scheme divides the whole geographic area into disjoint districts and all the cells within a district form a group. Now a CID does not only consist of a Cell ID, but also contains the Group ID this cell is part of. Therefore, information can now be stored in a vector which combines a group, with a BV containing all the cells inside of that group. This makes it so that the overhead for scope information is reduced compared to the BVC method. In this case the validity can first compare the group ID with the one associated with the data cache, before having to check specific data inside of the group. The downside to this method is that the group size matters, if the group gets too big the overhead for maintaining validity information is still very significant. While if it is too small, the chance of marking a valid data value as invalid is big, if it is outside of its original group.

The third and final method is ISI (Implicit Scope Information). Compared to the previous two methods which are both very simple, but might require a lot of bandwidth and cache memory to be used to store this information and perform the required validity checks, ISI attempts to reduce the size of the validity information, but also increases the complexity of the validation process. For this method the server enumerates the scope distribution of all items and numbers them sequentially, while the valid scopes within a scope distribution are also numbered sequentially. Now for each item the Scope Distribution Number and Scope Number within this distribution is stored, so that the information stored can be smaller than the previously mentioned methods, since only valid locations are stored, as compared to the previous where invalid locations are also stored.

3.3.4 Cache Replacement policies

Cache replacement policies base themselves on the associated location of data to determine the least useful unit (a victim) in a cache, which can then be discarded when room has to be made for new cache units. Two important examples of such locality are temporal locality and spatial locality. Temporal locality is the property that items which have been referenced recently are likely to be referenced again in the near future. Examples of this are LRU (Least Recently Used), where the least recently used cache units are deemed to be the least useful ones, and MRU (Most Recently Used) in which the most recently used units deemed to be the least useful ones. Spatial locality is the property that if an item has been referenced, other items that are physically located nearby in storage (or memory) are also likely to be referenced.

In [36] two cache replacement policies are described, where the first one's replacement value is based on the recency of use and in the second the replacement value is based on a distance function. The first cache replacement policy allows already existing replacement policies such as LRU and MRU to be used. To achieve this, when joining semantic regions, the region with the least value will have

to shrink when joined together with another region, which can be seen in figure 8. The second replacement policy uses a semantic distance, based on the Manhattan distance. The Manhattan distance is the “block distance” between the “center of gravity” of a region and the “center of gravity” of the most recent query, the lower the distance the closer the results of this region are linked to the specified query (e.g. two queries which require information about the same location on a map, these two queries would be linked closer to each other than two queries which are far apart). Thus regions with a value close to 0 are most important and are thus less likely to be discarded when free space is required. An example of this is figure 9, in which after executing the third query, Q2 has the biggest negative value. This means that when executing the next query, this query result has the most chance of getting removed (since it has the greatest distance to the most recently issued query).

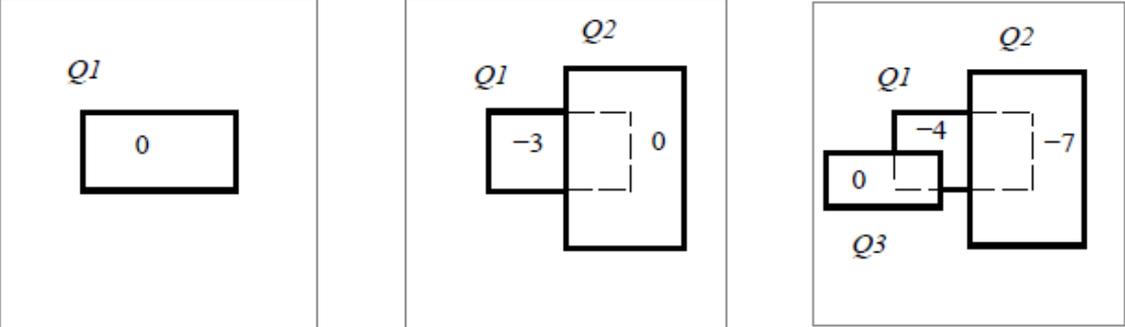
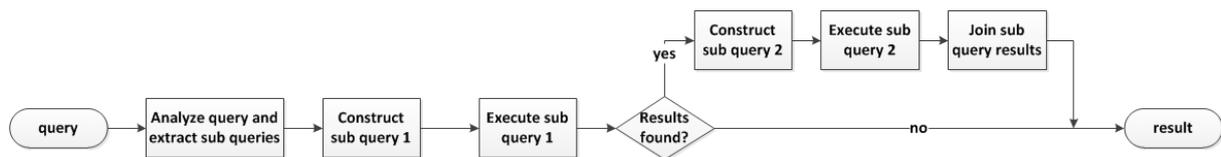


Fig. 9. Negative Manhattan distance between semantic regions and a query.

In this thesis the LRU policy is used, this is because of two reasons: 1) since results from queries which were added the longest time ago are the least useful for the execution of newer queries, 2) for simplicity.

Chapter 4. Query Pipelining

The purpose of this thesis is to investigate how the query mechanism can be improved, for use in the query service and the notification service. The first step in achieving this goal is to realize a query pipeline, which takes advantage of the fact that a Notification Service query mostly exists out of two sub queries that can be executed separately. The first being a query that always has to be executed on a given source, the second being a query that has to be executed on all the sources encountered before. This allows us to construct a mechanism that can take advantage of the fact that if the first query has no results, the second query does not have to be executed. Additionally, the results of the second query can be cached more effectively (query caching is elaborated in chapter 5). The general flow for this query pipeline is as follows:



This chapter explains how the query pipeline works. First we explain two mechanisms for determining these two query parts. After that we explain two strategies which can be used to construct these two new queries. Finally, we discuss a way to join these results, forming the same result sets one would receive when executing the query as a whole.

4.1 Query analysis

As mentioned before a query can be split up into two smaller queries, each having their own purpose. To form these two queries, the Notification Service query first needs to be analyzed in order to distinguish the *three* main parts of the query. An example of such a query can be found below. This query checks whether the newly encountered source contains information about a “region:shop” and calls this the eventEntity, based on this eventEntity it wants to know about all the sources it encountered before, which have something in common and that the user has visited in the a certain time frame up till now. First it needs to know what the query triggers are (e.g. a SCOUT eventEntity, a user), secondly it needs to know what the main parts for the first query are, and finally it needs to know what the final query has to contain. Next to these parts we also need to know what the shared variables are between the triple patterns. These shared variables can then be used for integrating the results in the final step.

```
SELECT ?shop ?item
WHERE
{
    ?user rdf:type em:User .
    ?shop rdf:type scout:eventEntity .
    ?shop rdf:type region:shop
    ?shop shop:sells ?item .
    ?item shop:relatedTo ?concept .
    ?user scout:wasNearby ?prevEntity .
    ?prevEntity scout:nearbyFrom ?time .
    FILTER ([now] - ?time < 172800)
    ?prevEntity shop:relatedTo ?concept .
}
```

SPARQL Query 5. Example query for analysis.

After performing the query analysis, we should be able to detect the following parts.

```
?user rdf:type em:User .
?shop rdf:type scout:eventEntity .
```

SPARQL Query 6. Example query analysis: triggers

```
?shop rdf:type region:shop
?shop shop:sells ?item .
?item shop:relatedTo ?concept .
```

SPARQL Query 7. Example query analysis: query part 1

```
?user scout:wasNearby ?prevEntity .
?prevEntity scout:nearbyFrom ?time .
FILTER ([now] - ?time < 172800)
?prevEntity shop:relatedTo ?concept .
```

SPARQL Query 8. Example query analysis: query part 2

To achieve this, two analysis mechanisms were created. The first one is automatic, meant for queries such as the one mentioned above; the second is a manual approach where hints have to be added to a query.

4.1.1 Automatic query analysis

The main idea behind the automatic query analysis approach is really simple; first we detect which of the query triple patterns belong to the trigger parts (see before). Once this is known, this information can be used to detect the first query part. After the first part has been detected, we can detect the second query part.

The automatic analysis assumes that the triple patterns present in the query are ordered in a certain way. For instance, it is assumed that the first two triples constitute the trigger part. Any triple that is linked to the resource of type “eventEntity” are considered to be part of the first query. Finally any triple not part of the trigger or first query are considered to make up the second query part. However, there are some issues that might arise, related to the ordering of the triples. If the triples are organized such as in the query example above where each part is followed by the next part, no problems exist. But there might be queries where the ordering of these parts is different, or triples from the different parts are mixed. An example of this can be found in the following query, where the ordering of the triples where the two query parts are mixed.

```
SELECT ?shop ?item
WHERE
{
    ?user rdf:type em:User .
    ?shop rdf:type scout:eventEntity .
    ?shop rdf:type region:shop
    ?item shop:relatedTo ?concept .
    ?prevEntity shop:relatedTo ?concept .
    ?shop shop:sells ?item .
    ?user scout:wasNearby ?prevEntity .
    ?prevEntity scout:nearbyFrom ?time .
    FILTER ([now] - ?time < 172800)
}
```

SPARQL Query 9. Example query with wrong order.

Since this issue prevents a fully automated solution from being applied in practice, rules have to be created which the query author must follow. This is discussed in the next subsection.

4.1.2 Query analysis using named graphs

As mentioned in the previous subsection, the author must follow some rules to make the query analysis behave accurately in every situation. More specifically, the author is required to employ named graphs in the query, in order to manually designate the different parts. Named Graphs is the idea that having multiple RDF graphs in a single document/repository and naming them with URIs provides useful additional functionality build on top of RDF. In our case it is used to specify which parts of the query belong to the triggers, which parts of the query belong to the first query part and which belong to the second query part. The following query is the same query as SPARQL Query 5, but uses named graphs. In this example, three named graphs are used (“http://wise.vub.ac.be/querypart/triggers/”, “http://wise.vub.ac.be/querypart/part1/” and “http://wise.vub.ac.be/querypart/part2/”), each denoting the different query parts.

```
SELECT ?shop ?item
FROM NAMED <http://wise.vub.ac.be/querypart/triggers/>
FROM NAMED <http://wise.vub.ac.be/querypart/part1/>
FROM NAMED <http://wise.vub.ac.be/querypart/part2/>
WHERE
{
  GRAPH <http://wise.vub.ac.be/querypart/triggers/>
  {
    ?user rdf:type em:User .
    ?shop rdf:type scout:eventEntity .
  }
  GRAPH <http://wise.vub.ac.be/querypart/part1/>
  {
    ?shop rdf:type region:shop .
    ?shop shop:sells ?item .
    ?item shop:relatedTo ?concept .
  }
  GRAPH <http://wise.vub.ac.be/querypart/part2/>
  {
    ?user scout:wasNearby ?prevEntity .
    ?prevEntity scout:nearbyFrom ?time .
    FILTER ([now] - ?time < 172800)
    ?prevEntity shop:relatedTo ?concept .
  }
}
```

SPARQL Query 10. Example query using named graphs.

It can be noted that this notation makes the query larger and more cumbersome to write; we could just as well rely on the author the order the triples in such a way that no problems arise. However, we feel this designation of triple patterns to query parts is more useful, as it makes the assignment more explicit, thus reducing the chance for errors.

4.2 Constructing the queries

Once the relevant parts of query are extracted (by means of the query analysis mentioned before), we need a way to create the two distinct queries, the first being the query that has to be executed on the encountered source (corresponding to the first query part), the second being the query that has to be executed on Environment Model (corresponding to the second query part).

To achieve this, two strategies (i.e. join and injection) are investigated, each with their own advantages and disadvantages.

4.2.1 Join strategy

The join strategy is the simpler of the two strategies. It constructs the two queries “as is”, not taking into account the results of the other query. The two queries below are an example of this strategy.

```
SELECT ?shop ?item
WHERE
{
    ?user rdf:type em:User .
    ?shop rdf:type scout:eventEntity .
    ?shop rdf:type region:shop
    ?shop shop:sells ?item .
    ?item shop:relatedTo ?concept .
}
```

SPARQL Query 11. Example of query 1 after construction.

```
SELECT ?shop ?item
WHERE
{
    ?user rdf:type em:User .
    ?prevEntity shop:relatedTo ?concept .
    ?user scout:wasNearby ?prevEntity .
    ?prevEntity scout:nearbyFrom ?time .
    FILTER ([now] - ?time < 172800)
}
```

SPARQL Query 12. Example of query 2 after construction.

The advantage of this is that the second query, which is executed on the Environment Model, contains all the results related to this query, and not only the ones related to the results of the first query. This can be useful for caching, since the more general result can be reused for more queries, this is discussed in the next chapter. But the downside is that the results of both queries need to be combined afterwards, and more resources have to be used in the execution of the query. The extra resource usage comes from the extra results that have to be checked and combines, which requires more cpu usage and memory.

4.2.2 Injection strategy

Since the previous strategy might be inefficient, a second strategy has been investigated which takes advantage of the fact that the first query is always executed first. Consequently its results will be available by the time the second query has to be executed. More specifically, a filter, based on the shared variables in both sub queries, is added to the second sub query, so only results relevant to the results found for the first query are returned. This should reduce resource usage in the query execution (in case the number of results for the first query is limited), and avoids having to combine the results of both parts afterwards (or at least reduce the time and memory needed to combine these results). However, a major downside to this approach is that it reduces the amount of other queries where this result can be used in, making it so that results can be reused in fewer situations if caching gets introduced. An example of the second sub query in this case can be found below (SPARQL Query 13; the first sub query remains the same as before). Generally, it looks the same as the query shown in SPARQL Query 12; apart from an extra filter, which takes into account the results from the first sub query.

```

SELECT ?entity ?item
WHERE
{
    ?user rdf:type em:User .
    ?prevEntity shop:relatedTo ?concept .
    ?user scout:wasNearby ?prevEntity .
    ?prevEntity scout:nearbyFrom ?time .
    FILTER ([now] - ?time < 172800)
    FILTER((regex(str(?concept) , "^http://example.com/concept1$") ) || (regex(str(?concept) ,
    "^http://example.com/concept2$") ))
}

```

SPARQL Query 13. Example of sub query 2 after injection.

4.3 Joining the results

The final method used in the query pipeline is the one used to join results from both queries. This method is simple and falls back into two cases. The first case occurs when the two queries have shared variables, the second when they have no shared variables. Since a given query might not always contain the shared variables in its SELECT variables, all variables are added to the SELECT clause in the two generated queries. This is done to enable the joining of the results on the values of the shared variables afterwards, and to allow more queries to be able to reuse the results when caching gets introduced in the next chapter.

In the first case, every row from the first result needs to be joined with every row from the second result, based on the values of the shared variables. An example of this can be found in result Table 3, where the results found in Table 1 and Table 2 are joined. As can be seen in the example, since the “http://example.com/concept2” value of the “?concept” shared variable in the first query result is not contained in the second result, this concept does not appear in the final result.

?shop	?item	?concept	?user
http://example.com/shop1	http://example.com/item1	http://example.com/concept1	Scout:user
http://example.com/shop1	http://example.com/item2	http://example.com/concept2	Scout:user
http://example.com/shop1	http://example.com/item2	http://example.com/concept1	Scout:user
http://example.com/shop1	http://example.com/item3	http://example.com/concept3	Scout:user
http://example.com/shop1	http://example.com/item3	http://example.com/concept1	Scout:user

Table 1. Example output SPARQL Query 11.

?concept	?prevEntity	?user	?time
http://example.com/concept1	http://example.com/item22	Scout:user	500
http://example.com/concept3	http://example.com/item33	Scout:user	550

Table 2. Example output SPARQL Query 12.

?shop	?item	?concept	?prevEntity	?user	?time
http://example.com/shop1	http://example.com/item1	http://example.com/concept1	http://example.com/item22	Scout:user	500
http://example.com/shop1	http://example.com/item2	http://example.com/concept1	http://example.com/item22	Scout:user	500
http://example.com/shop1	http://example.com/item3	http://example.com/concept1	http://example.com/item33	Scout:user	500
http://example.com/shop1	http://example.com/item3	http://example.com/concept3	http://example.com/item33	Scout:user	550

Table 3. Example output SPARQL Query 11 and 12 after joining.

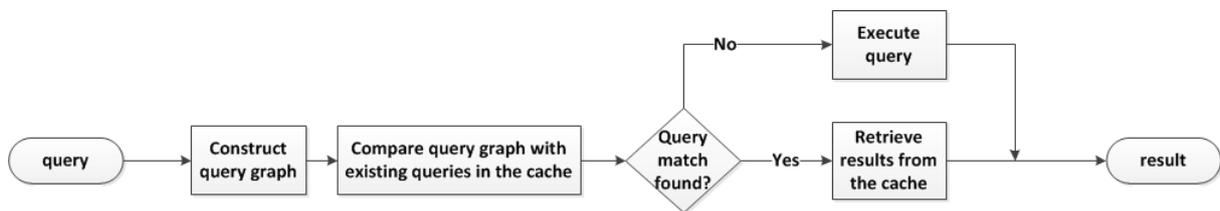
The second case is simpler, and occurs where there are no shared variables between the two query results. This means that every result row from the first query needs to be combined with every results row from the second query (resulting in a Cartesian product). An example of these results, when not taking into account that “?concept” and “?user” are shared variables, can be seen in Table 4.

?shop	?item	?concept	?prevEntity	?user	?time
http://example.com/shop1	http://example.com/item1	http://example.com/concept1	http://example.com/item22	Scout:user	500
http://example.com/shop1	http://example.com/item2	http://example.com/concept2	http://example.com/item22	Scout:user	500
http://example.com/shop1	http://example.com/item2	http://example.com/concept1	http://example.com/item22	Scout:user	500
http://example.com/shop1	http://example.com/item3	http://example.com/concept3	http://example.com/item22	Scout:user	500
http://example.com/shop1	http://example.com/item3	http://example.com/concept1	http://example.com/item22	Scout:user	500
http://example.com/shop1	http://example.com/item1	http://example.com/concept1	http://example.com/item33	Scout:user	550
http://example.com/shop1	http://example.com/item2	http://example.com/concept2	http://example.com/item33	Scout:user	550
http://example.com/shop1	http://example.com/item2	http://example.com/concept1	http://example.com/item33	Scout:user	550
http://example.com/shop1	http://example.com/item3	http://example.com/concept3	http://example.com/item33	Scout:user	550
http://example.com/shop1	http://example.com/item3	http://example.com/concept1	http://example.com/item33	Scout:user	550

Table 4. Example query result after joining with no shared variables.

Chapter 5. Cache

As mentioned before, one of the purposes of this thesis is to investigate how the query mechanism, which is used in the Query Service and Notification Service, can be improved. Chapter 4 discussed the first step in achieving this goal, by introducing a query pipeline for Notification Service queries and explaining how it can improve the query execution performance for that. This chapter explains the second step, namely query caching. The caching of Notification Service query results is made possible by caching only the second part of the query (which is executed on the Environment Model, i.e. previously encountered sources). Because of this, these results have a high chance of being used across multiple queries. In contrast caching the results of a full query would have no use, since the first part has to be executed on the newly encountered source so that the query would have to be executed again completely (i.e. part 1 and part 2) and would nullify the use of the cache. The general flow for the query execution mechanism:



First, query caching is explained, what it does and how it can help. Secondly, a graph construction mechanism is explained, after which a way to compare two graphs from different queries will be discussed. Finally, we elaborate on the employed caching eviction strategy.

5.1 Query caching

As mentioned before in section 3.3.1, there are four main caching strategies that can be used on mobile devices. The first being source caching, followed by page caching, tuple caching and query caching (Semantic caching). This section will explain what these four caching strategies do and how query caching can be used to improve performance.

In this thesis we only discuss query caching, which will be used on the sub queries created by the pipeline in the previous chapter. Since the results of these sub queries have higher chance of being reused in other queries, then the results of a full query. Currently, some of the more advanced features (i.e. retrieving the remaining results for a given sub query) are not needed yet, leaving these open for future work, which is discussed in chapter 8. The following sections will discuss our implementation of this query caching, starting with the construction of query graphs, how to compare them and finally how the actual caching strategy is formed.

5.2 Query graph construction

In order to reuse query results from previous queries, we need to know whether the current query corresponds to the one of the caches queries. A first step in this mechanism is to create a high-level representation of the query in the form of a graph. Such a query graph encodes the underlying graph structure of the SPARQL query, and allows us to deal with so-called syntactic variants of queries (e.g. queries that are structurally the same but use different variables names, or use a different ordering for the triple patterns).

Since the underlying structure of a SPARQL query already is a graph structure, converting such a query to a graph that can be used programmatically is fairly simple. For each unique subject and object available in a query, a node is created together with some meta-information (such as a name and a type), and the predicates found in the query are used as edges between these nodes.

This thesis also investigates semantic relations, such as inverse relations (since a query using inverse relations could end up returning the same results as a query using normal relationships), in order to improve the graph comparison process. This allows us to deal with queries that use different predicates denoting the same concept (for instance, predicates that are the inverse of each other), and therefore yield the same results. In order to deal with inverse relations, we decided to create a graph that, apart from the “regular” predicates between a node A and a node B, also contains the corresponding inverse predicates between node B and node A. This way a query can be compared to both a query using the same relations and a query using inverse relations, which are retrieved from a helper class (currently only inverse relationships which are added manually can be used, but future work can extend this class to allow automatic retrieval using the RDFS definition of these relations). An example of such a query graph can be seen in figure 12, which is a graph representation for SPARQL Query 11. The black edges represent the edges which are defined in the query, while the orange edges are the inverse edges which have been added for better comparison.

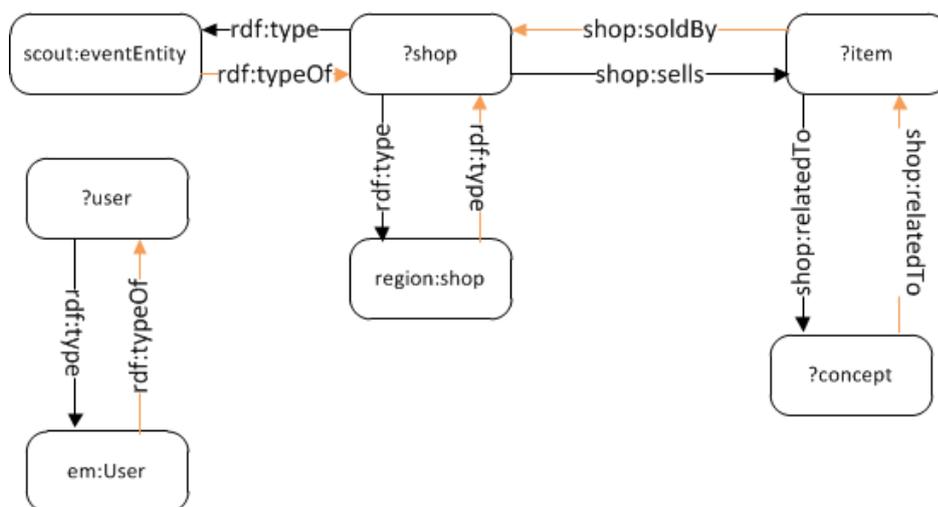


Fig. 10. Example query graph 1.

5.3 Query graph comparison

To know whether a cached query result can be re-used for a given query, the query graph associated with the cached result and the given query graph need to be compared. In case there is an exact match between the two query graphs, or the given query is a subgraph of the cached query graph, the cached query result may be re-used. This section states the conditions or rules to which both query results must adhere, before the cached query results can be re-used to answer the given query (graph A represents the new query being executed, and graph B is the query stored in the cache):

- The two graphs need to contain the same amount of nodes and edges. If graph B would contain more nodes or edges, this would mean that there are more restrictions in place, and the result set would be smaller than the one otherwise generated by graph A. On the other hand, if graph B would contain less nodes or edges, this would mean that there are less

restrictions, causing the result set to be bigger than the one otherwise created by graph A. Although a result that is bigger might contain the results for graph A, which can be filtered out, not enough information is stored in graph B to allow this filtering to occur. Future work can be used to improve this and is discussed in chapter 8.

- For every node in graph A, there has to be a node available in graph B that has the same relationships (or predicates). If a node cannot be found in graph B that is a proper match for a node in graph A, this means that there is at least one node that contains more restrictions. Because of the previous rule this is not allowed, so that the two graphs are not the same.
- While the number of nodes and edges of graph A and B need to be the same, graph A is still allowed to be a (query) subgraph of graph B. This can for instance occur when graph A contains a node A1, which is for instance a Literal (type), and graph B contains node B1 but is a variable (type), while both node A1 and B1 are in the same relative “position” and contain the same inbound and outbound edges. In this case the results for query graph A are contained within the results of B. At the same time, the required information is present in the results of B to filter out the results in order to obtain the results for A; since the node B1 is a variable, it will most certainly include the Literal which is represented by node A1. A downside to this, which can be grounds for future work, is that there might be too many results available in B, making the filtering process expensive. Note that B may not be a subgraph of A (i.e. the other way around); because of the extra restrictions in graph B some of the results for query graph A would be discarded. This also applies to subclasses and sub properties, which might be part of graph A, which can be retrieved from the same helper class mentioned in section 5.2, the same can be said for edges, which might also be variables in some queries.

In case a cached query graph B and a given query graph A comply with the above rules, the cached query results can be reused to serve the given query. In figure 13, an example of two such graphs can be found, the one on the left is the same as seen in figure 12, which is based on SPARQL Query 11. On the right we see the same graph, where one of the triples has been altered (“?shop shop:sells ?item .” was replaced by “?item shop:soldBy ?shop .”)

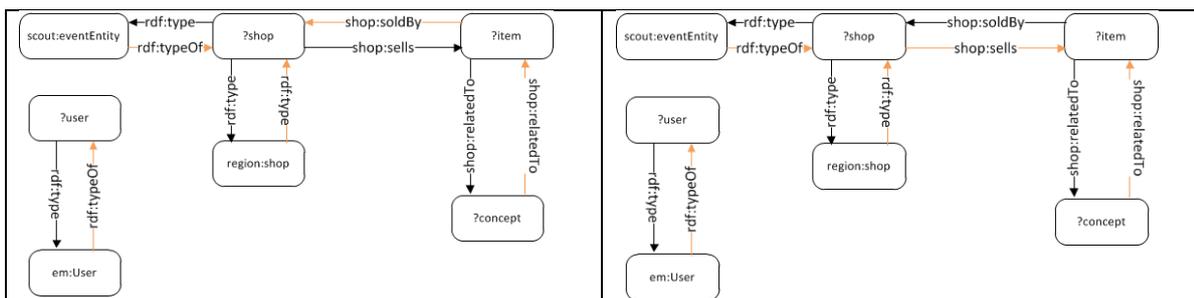


Fig. 11. Example of identical query graphs.

Apart from triples which form the basis for the comparison mentioned above, there are also advanced forms of nodes, such as the FILTER clause available in many SPARQL Queries. These advanced forms aren’t discussed in this thesis, and are further explained in chapter 8 (Future Work).

5.3.1 Graph traversal

This section will describe the graph traversal algorithm used to compare two graphs. First pseudo code is given, which is explained afterwards in the following paragraphs.

```

1  FOR all nodes NA in graph A
2      IF CALL findPathFromNode with NA == false THEN
3          RETURN -1
4      END
5  ENDFOR
6  RETURN 0
7
8  findPathFromNode: (IN: node NA; OUT: true or false )
9      IF NA is already mapped to a node in graph B THEN
10         RETURN true
11     ELSE
12         FOR all nodes NB in graph B
13             IF CALL traverse with NA, NB == true THEN
14                 RETURN true
15             END
16         ENDFOR
17         RETURN false
18     END
19
20  traverse: (IN: node NA, NB, PNA, PNB; OUT: true or false)
21      IF NA == PNA THEN
22          IF NB == PNB THEN
23              RETURN true
24          ELSE
25              RETURN false
26          END
27      END
28      IF NA already found, but not processed THEN
29          RETURN true
30      END
31      IF NA is already mapped to a node in graph B THEN
32          IF this node == NB THEN
33              RETURN true
34          ELSE
35              RETURN false
36          END
37      END
38      IF #edges(NA) == #edges(NB) THEN
39          IF NB not mapped to a node in graph A THEN
40              IF NA similar to NB THEN
41                  IF CALL compare_edges with edges(NA), edges(NB), NA,
42                     NB == true THEN
43                      MAP NA to NB
44                      RETURN true
45                  END
46              END
47          END
48      END
49      RETURN false
50
51  compare_edges: (IN: edges(NA), edges(NB), NA, NB; OUT: true or false)
52      FOR all edges EA in edges(NA)
53          FOR all edges EB in edges(NB)
54              IF EA is similar to NB THEN
55                  IF CALL traverse with to node from EA, to node from
56                     EB, NA, NB == true)
57                      RETURN true
58              END
59          END
60      ENDFOR
61  ENDFOR
62  RETURN false
63

```

The comparison of two query graphs happens as follows. For two graphs A and B, we check whether A is the same graph as B or A is a sub graph of B. This is done by trying to map every node in A to a

node in B. First some checks are done to see if graph B is bigger or smaller than graph A (i.e. does not have the same amount of nodes or edges); if it does the comparison stops right away, since the rules mentioned before say that this is not allowed; if it does not we continue with the algorithm mentioned in the pseudo code above.

First, we start by traversing every node (NA) in graph A. For each of these nodes we try to find a node (NB) in graph B, by calling the findPathFromNode procedure. This procedure will first check if we have not found a mapping for this node yet, while we were checking a previous node; if there is a mapping already, we can stop right away and continue with the next node in A; if there is no mapping yet we continue by traversing over every node (NB) in graph B. For each of these nodes, we call traverse, which takes a source node and a target node and compares these with each other. In this procedure, we first check if there is no self-loop, by checking if the current node NA is not the same as the previous NA node called PNA; if it is, we compare NB with its previous node PNB. If this also contains a self-loop then we found our node, otherwise we did not find a match. If NA is not the same as PNA we continue by checking if we are already processing node NA, to prevent loops from occurring. If we are already processing this node, we stop and return true. If we are not, we continue by checking if NA is already mapped to a node in graph B. If NA is mapped to a node in graph B, we check if this node is NB and return true in case they are the same, otherwise we return false and continue by checking the next node in B.

Once we know that NA is not being processed yet and that it has no existing mapping to a node in B, we continue by checking if NA and NB have the same amount of edges. If they do not we return false and continue with the next node available in B. If they do we continue by checking if NB is not mapped to a node in A yet. If it is we return false and continue with the next node in B. If it is not, we check if NA and NB are similar (e.g. NA and NB are both variables, NA is a literal and NB is a variable, NA and NB are both literals and have the same contents). If they are not similar we return false and continue with the next node in graph B. If they are similar we continue by comparing their edges in the procedure compare_edges. Compare_edges will try to match each edge starting from node NA to an edge starting from node NB and does this by checking if the two edges are similar, after which it continues by comparing the two “to nodes” using the traverse procedure again. If compare_edges returns true, we have found a match after which we map NB to NA and return true, thus saying we have found a match for this node.

Once we have found a match for each node in graph A to a node in graph B, we can say that these two graphs are similar.

5.4 Eviction Strategy

The previous sections discussed which type of data is cached, and how queries can be compared. This section discusses the eviction strategy chosen to decide what elements are allowed to remain in the cache and which should be removed when extra space is required. A couple of existing, and commonly used, eviction strategies were previously mentioned in chapter 2. These eviction strategies were First In First Out (FIFO), Least Recently Used (LRU) and Least Frequently Used (LFU), and the use of any of these three depends on the expected access pattern.

Since the order queries are executed in the SCOUT framework is not predetermined, but queries can be executed in any given order, based on the information the user needs and at any given time, when the user needs certain information. The First In First Out eviction strategy can be ruled out being a

suitable strategy, since whenever room is needed the “oldest” query result (i.e. the first one to be executed) would be removed, even if this query might be executed more than any of the other queries.

Both Least Recently Used (LRU) and Least Frequently Used (LFU) could be suitable approaches. However, since LFU requires a certain overhead to store the amount of times a certain query has been used, LRU which has a minimal overhead and is a fast algorithm, is used. Another reason for this choice is that there already exists an implementation of this strategy in the SCOUT framework.

To prevent unused data from using up memory, the SCOUT framework LRU implementation is optimized to remove cache elements that have not been used for a certain amount of time. So cache elements which are out of date, are already removed before new elements get added, allowing the search process to be improved, since there are less elements available in the cache. To achieve this, a timestamp is stored for each element containing the time the element was last accessed in the cache.

The following subsections explain the various operations that have to be performed by the cache, such as adding an element, updating an existing element and requesting an element from the cache. Since our cache contains query results, and we want to be able to compare the query associated with every query result to a given query, this query is used as key for its associated query result.

5.4.1 Add an element to the cache

In order to free up space before trying to add an element to the cache, the optimization mentioned before, which uses a timestamp for each elements, will remove any elements out of the cache which have expired (meaning they have not been used in a certain time), which speeds up the lookup process when an element has to be found, since fewer queries have to be compared.

Before an element can be added to the cache, three conditions are checked. The first condition checks whether the cache already contains this key, by using graph comparison; if it does the update method is called instead to prevent adding the same element multiple times. Afterwards, the second condition will check if the element fits into the cache to prevent removing (useful) information when it is not needed (since the new element will not fit in the cache even after removing elements). And finally, the third condition checks whether there is enough room available in the cache for the new element; if there is not enough room, our eviction strategy is used to evict the Least Recently Used elements. Once enough space is made available the element is added to the cache, and receives a timestamp.

5.4.2 Update an element in the cache

Whenever a new source is detected, the Source Index Model will be used to check whether this source contains relevant information for any of the queries available in the cache. If it does, the entire query will be executed again on the Environment Model (i.e. all relevant encountered sources) and the result in will be updated in the cache using the workflow mentioned in the following paragraph.

The workflow of updating an element in the cache is similar to the one mentioned in the previous subsection (i.e. when adding elements to the cache). First, the optimization mentioned before will remove any elements from of the cache which have not been used for a long time. Note that it is possible that, during this process, the element, that was about to be updated, is removed. In that

case, the element will not be re-added to the cache, since as mentioned before, the timestamp on an element is only updated when adding it to the cache, or when an element is requested.

Before we can update an element in the cache, the three conditions mentioned above are checked. The first condition, which checks whether an element is still available in the cache, is used to determine if the element is still available to be updated. The second condition is used to check whether the element is larger than the allowed cache size; if not, the existing element is updated; else, the element is removed from the cache, since its new size would exceed the max cache size. And finally, the third condition checks whether there is enough room left in the cache, and otherwise uses our eviction strategy to remove less Least Recently Used element(s). Because of this, it might happen that there still is not enough room available for this element to be updated.

5.4.3 Find an element in the cache

As opposed to the previously mentioned operations, our optimization of removing “unused” data is not performed when looking for an element in the cache. This step is not done, because the content of the cache is not changed during this operation and no room has to be made for adding or updating an element. This allows the search process to be optimized, at the cost of the add and update operations being slower. If the element is found, then its timestamp is updated to make it the most recently used element.

Chapter 6. Implementation

Two mechanisms are developed: 1) a query pipeline mechanism, and 2) a query caching mechanism to optimize the query resolving, and reduce the amount of information that has to be downloaded when queries are posed to the Environment Model.

This chapter explains the concrete implementation of these mechanisms in detail. First, an overall view of the implementation is given describing how these two mechanisms can be used in combination to achieve efficient query resolving in the Query and Notification Service. Next, the rest of the sections describe each component in detail. Section 6.2 discusses query pipelining using the two methods mentioned before, namely the join and injection strategy. Section 6.3 discusses how queries are compared, while section 6.4 discusses how the caching mechanism works.

6.1 Overview

Previously, whenever the Notification Service had to execute a query after encountering a new source, this query had to be executed as a whole on both the encountered source as well as the Environment Model. This thesis investigates how notification service queries can be constructed and executed in such a way that the execution performance is increased and results can be cached more efficiently. Furthermore, the efficient caching of the results of pattern-matching queries (e.g. such as SPARQL) is investigated.

The first step of the proposed approach is splitting up Notification Service queries into two sub queries which can be fired separately on either the encountered source or the Environment Model. One of those sub queries is the part specific for the given source, the other sub query is the part which has to be executed on previously encountered sources (or Environment Model). The purpose of this is adding support for the caching of the second query part, since this sub query has more chance of being reused in other queries, or being executed multiple times in a short period of time. Whenever the first sub query has no results, the second (more expensive query) does not have to be executed anymore, making sure no valuable resources are spent.

As a second step, an efficient query caching technique is proposed in which SPARQL query results from the Environment Model (which runs the query on previously encountered sources) can be stored and possibly updated if new sources are encountered.

When taking into account the layered architecture of the SCOUT framework, this thesis mainly focuses on the Environment Layer, more specifically the Query and Notification Service components.

The following subsections explain how these two mechanisms can be integrated into the SCOUT framework and how they can work together to improve performance.

6.1.1 Integration in the SCOUT framework

As mentioned before, the Environment layer stores and integrates data about the user and his current environment and provides services to obtain information from nearby Web presences. The two major contributors to this are the query service and the notification service which provides the applications a way to 1) query for information about the user and his environment and 2) receive notifications about certain events based conditions encapsulated as queries. Since both of these services thus heavily rely on queries to be executed, improving this system to use cached query results, where possible, can lead to an improved performance of the system.

In most cases, queries executed by the Notification Service, consist of two parts. The first part has to be executed on the encountered source every time, while the second part has to be executed on previously encountered sources. Therefore, a system can be created which identifies these parts and then executes them using a query pipeline. This means that we can first execute the first part on the encountered source and if this provides results, execute the second part (e.g. by injecting results from the first part into the query for the second). To further improve the performance of this system, the results of the second sub query can be cached using the aforementioned caching mechanism, which must then be kept up to date when new sources are encountered.

6.1.2 Encountering a new source

In a mobile environment, which the SCOUT framework was created for, the context and environment are constantly changing, encountering new sources which can be used to provide more information about the environment to applications running on top of the framework.

When encountering a new source, RDF information is provided to both the Environment Model and the Relation Model. The Environment Model then sends a notification to the Source Index Model, which is used to extract information about the resources available in a source and builds an index. Later on this index can be used to determine which sources provide relevant information for a given query, its cache and the Query Result Cache, that together with the help of the Source Index Model updates the results in the cache if this new sources provides relevant information. An example of this can be seen in figure 10.

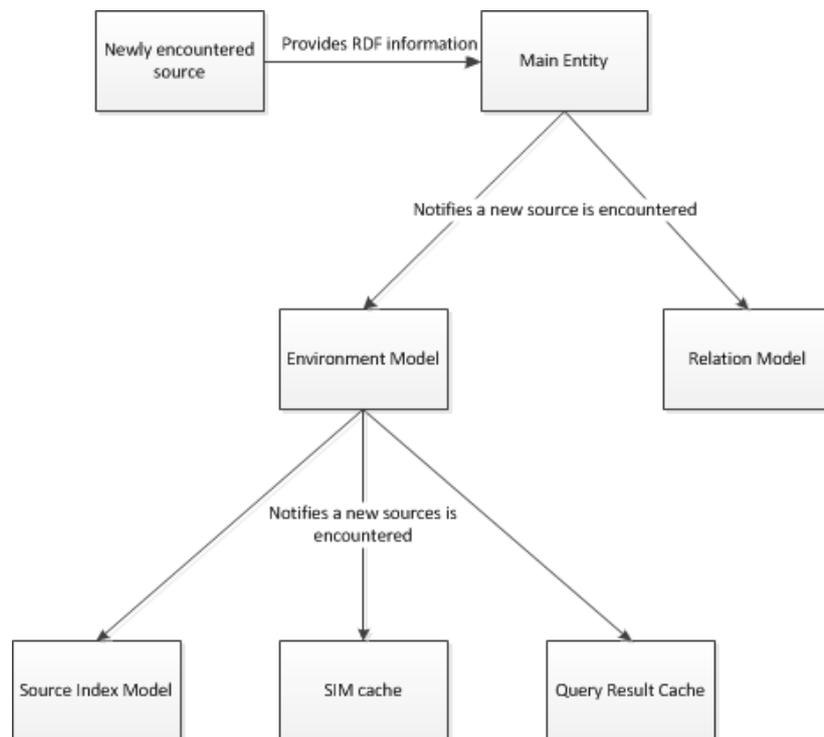


Fig. 12. Notification system of newly encountered sources.

6.1.3 Executing a query

Whenever a query has to be executed by the Notification Service, the system checks whether it can detect the two query parts. If both query parts are detected the Notification Service executes the first part of the query; subsequently, it can either retrieve results for the second part out of the

cache, or it executes the second part on the Environment Model (and then adds this result to the cache). A simple example of this can be seen in figure 11.

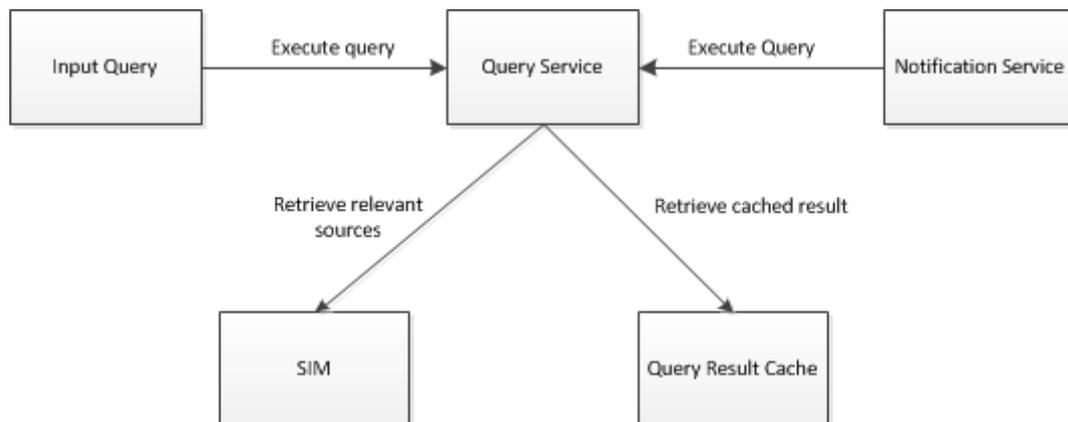


Fig. 13. Simple Query Execution Example

6.2 Query Pipeline

A first step to optimize query execution in the Notification Service is to provide a query pipeline mechanism, which is able to extract distinct sub queries out of a given query. These sub queries can then be executed separately, reducing resource usage where possible. In a second step, they allow the introduction of the second query optimization in the Notification Service, namely the query caching mechanism (see section 6.4).

More specifically, this pipelining mechanism allows us to discard the execution of the second sub query if the first one did not provide any results, and so reducing the amount of information that has to be downloaded. Chapter 4 explained the concept of this query pipeline, first by explaining two strategies used for analyzing a query and extracting the relevant information, after which two query construction strategies were discussed. The first strategy, being the join strategy, executes both sub queries as they were found in the original query, and the second strategy, being the injection strategy, adds a filter to the second sub query based on the results from the first. At the end a strategy was discussed, which is used to combine the results from the two sub queries, forming the actual result of the original query. The following subsections discuss each of these mechanisms separately, explaining how each strategy was implemented.

6.2.1 Splitting up the query

This section discusses the query analyzing mechanism used to extract the required query parts for later query execution. Since the automatic query analysis strategy had some problems (as mentioned in section 4.1.1.), only the implementation of the manual query analysis strategy is explained here.

The query analysis mechanism is a strategy that generates a QueryPartData model. This model can then be used later on (see the next subsection) for constructing the two queries. To create this QueryPartData model, a QueryPartComposer is employed which uses 1) the SPARQLParser class to parse the query into an AST¹⁶, and 2) an implementation of a query Visitor class, called the QueryPartVisitor, to visit the construct AST and extract the required data from the query. Both of

¹⁶ Abstract Syntax Tree

these are classes from the SPARQL Engine for Java¹⁷. The employed query analysis technique depends on the concrete QueryPartVisitor subclass used by the QueryPartComposer. For instance, the aforementioned manual query analysis strategy is implemented by the QueryPartVisitor2 class. Figure 14 shows the class diagram for these components.

The QueryPartModel has seven methods which can be used to retrieve the relevant query information.

- getSelectionVars: returns all the selections variables which are used in the given query.
- getTriggers: returns list of triggers (triple patterns) extracted from the given query.
- getPart1: returns a list of all elements available for use in the first sub query.
- getPart2: returns a list of all elements (e.g. triple patterns, union clauses, filters) available for use in the second sub query.
- getNameSpaces: returns a HashTable containing all the namespaces used in original query.
- getShared: returns a list of shared variables between the first and second sub query.
- isValid: returns whether or not the different query parts could be extracted from the given query.

The QueryPartComposer has one method:

- getManual: based on a given query, this method applies the QueryPartVisitor2 associated with the second query analysis strategy (i.e. the manual strategy).

The QueryPartVisitor is actually a subclass of the PrefixQueryVisitor class which is used to extract the namespaces from the query and itself implements the basic Visitor class from the SPARQL Engine library (this class was already available in the SCOUT framework). The QueryPartVisitor class inherits all methods from this superclass, and additionally contains the same seven methods available in the QueryPartData model, which are used by the QueryPartComposer to populate the QueryPartData model (see QueryPartModel for more information).

Since a QueryPartVisitor contains various functions that are not used by our strategy, only the ones implemented for use by our employed query analysis strategy (QueryPartVisitor2) are explained here.

- visit(ASTFilterConstraint): extracts a FILTER node from the given query.
- visit(ASTTripleSet): extracts the triple patterns present in a query.
- visit(ASTGraph): determines which part the following nodes (i.e. visited after this method invocations) belong to, based on the named graph visited in this method.
- visit(ASTUnionConstraint): detects UNION graphs in the query, and extracts the relevant elements inside these unions. Note, although there is a visit method for triples, we can only extract these here, since the triple pattern visit method is executed before this one, meaning there is no way to link these triples to this UNION node.

Next to the methods inherited from the QueryPartVisitor, the following two methods are also available:

¹⁷ <http://sparql.sourceforge.net>

- AddNode: adds the node (i.e. triple pattern, FILTER, UNION clause) to the correct query part (trigger, part1 or part 2), based on the information provided by the named graph visited before.
- ProcessNode: based on the given unboundStatement, extracts the object and subject data, which are then stored, to be able to detect which of these variables are shared between the two query parts.

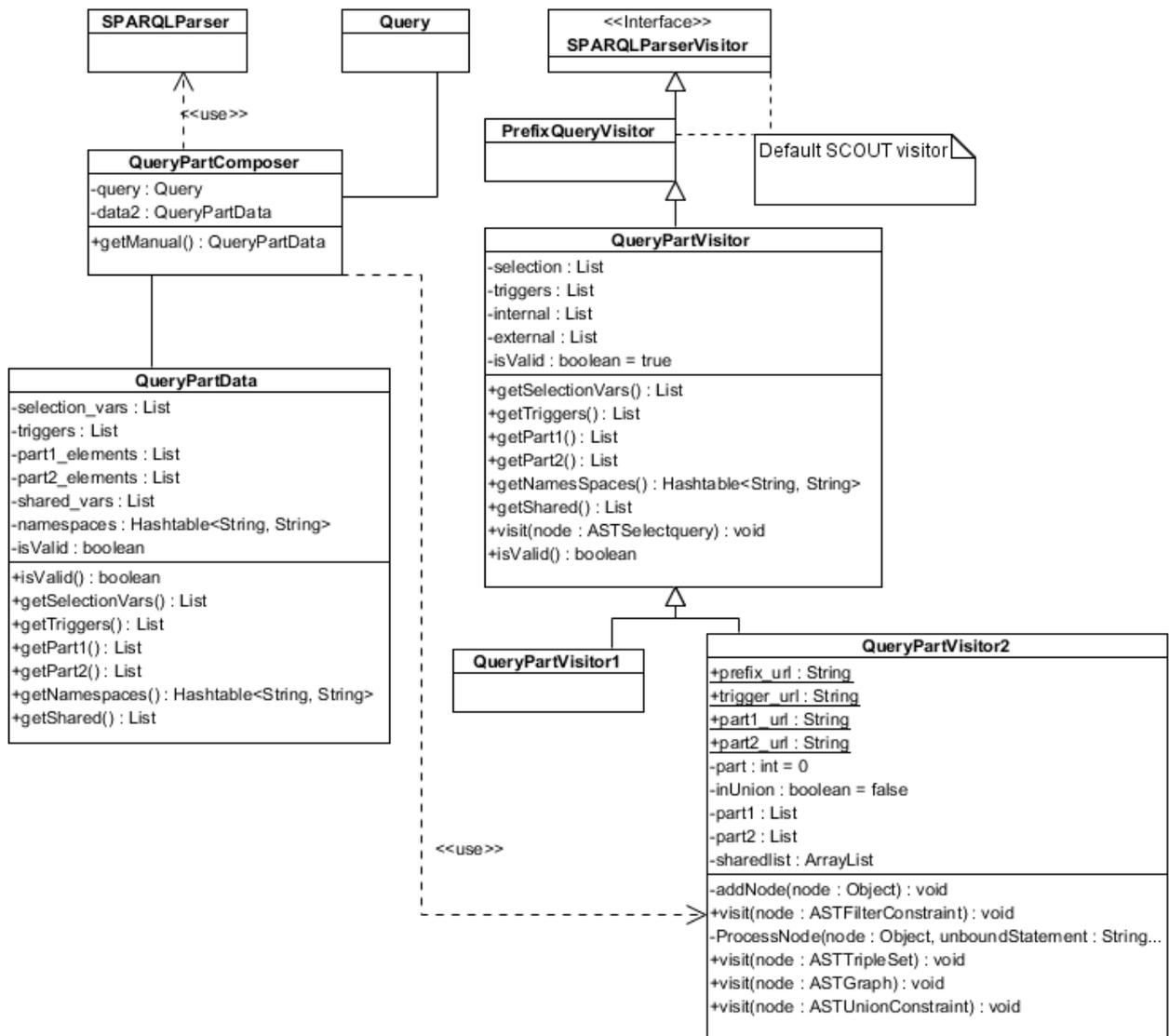


Fig. 14. Query Analysis Class Diagram.

6.2.2 Query Construction and execution

This subsection explains the two strategies used to create the new queries after the relevant parts have been extracted during the query analysis. Since the Query class available in SCOUT did not have support for generating WHERE clauses, a small modification to this class was made. Furthermore, a subclass of the Query class (called Query2) was added, to add support for adding separate query elements, such as triple patterns, UNION clauses, to a query. Another reason for this was the dependency on the SPARQL Engine Library to be able to process these query elements, which is not

needed for regular queries (i.e. using the Query class). Figure 15 shows the class diagram for this new class.

The Query class now has one extra method:

- generateWhereClause: returns the WHERE clause for the query, which can be extended by subclasses to generate their own WHERE clause.

The Query2 class keeps one field:

- nodes: a list containing the elements (generated by the Visitor from the SPARQL Engine library) required to build the WHERE clause in this query.

Next to the methods inherited from the Query class, the following methods are provided:

- AddNodes: adds the elements found in a list (e.g. the list containing elements for part 1) to the nodes list for later use in the query.
- AddNode: adds a single element to the query.
- generateStringForType: based on the type of element (triple pattern, UNION, ...), creates a string representation for use in the WHERE clause.

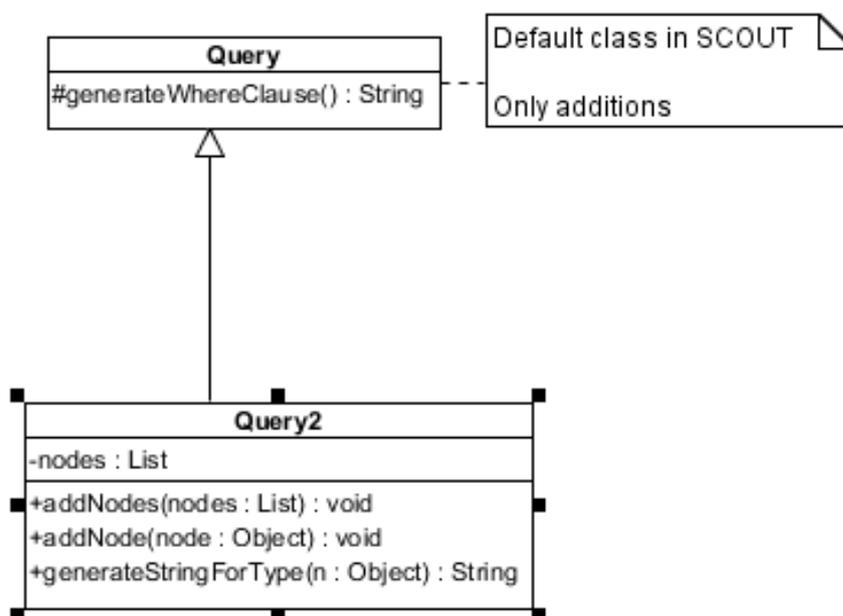


Fig. 15. Query2 Class Diagram.

6.2.2.1 Join Strategy

The join strategy makes use of two steps to construct and execute the required queries. Firstly, the first sub query is constructed by creating an instance of Query2, providing it with the associated namespaces, triggers and part1 elements. After finishing the first sub query, the query is executed; depending on the results of this query, the entire query process is stopped (no results found), or the next step of the query pipeline is executed. This second step involves creating a second instance of Query2, again providing it with the associated namespaces, triggers and part2 elements. Then the second sub query is executed after which the results are joined (as explained in section 6.2.3). Note

that parallel execution of both queries is not supported. This is because, in our setting, we need to avoid the execution of the second query as much as possible, depending on whether or not the first query has any results (see section 6.1).

6.2.2.2 Injection Strategy

The injection strategy is fairly similar to the join strategy mentioned in the previous subsection. Firstly, it creates the first sub query in the same way and executes it. Depending on whether the results are available, the pipeline either stops or continues. The difference with the join strategy is that the second query receives an extra element, namely a FILTER node, which is created based on the values of the shared variables from the results of the first query. Then, the sub query is executed and the results can be joined using the method described in subsection 6.2.3 to add the extra values which might be returned in the first query, but are not part of the shared variables added to the FILTER clause.

6.2.3 Joining the results

After having executed both sub queries, the results have to be joined together (see section 4.3). To achieve this QueryJoinResult was implemented, which specializes the SimpleQueryResult class. Figure 16 shows the class diagram for this class.

QueryJoinResult adds three methods to the ones inherited from SimpleQueryResult:

- **Combine:** used internally as a more optimized solution to combine the result sets of two queries if no shared variables are present.
- **Join:** responsible for the combining of results in case shared variables are present. This combining is based on the HashJoin algorithm. This algorithm uses the following steps to join results: first, a HashMap is created for the smallest result set, which indexes on shared variable values and keeps as values the lists sharing the same variable values. Secondly, the largest result set is traversed, creating a hash based on the shared variable values and checking whether this hash can be found in the HashMap. If a match is found, the results are combined into one or more new result rows (implemented by QueryJoinResult instances), depending on the amount of rows found in the HashMap value list for that specific key.
- **AddValueToRow:** used internally to add the results to a result row, taking into account the order of the selection variables found in the original query.

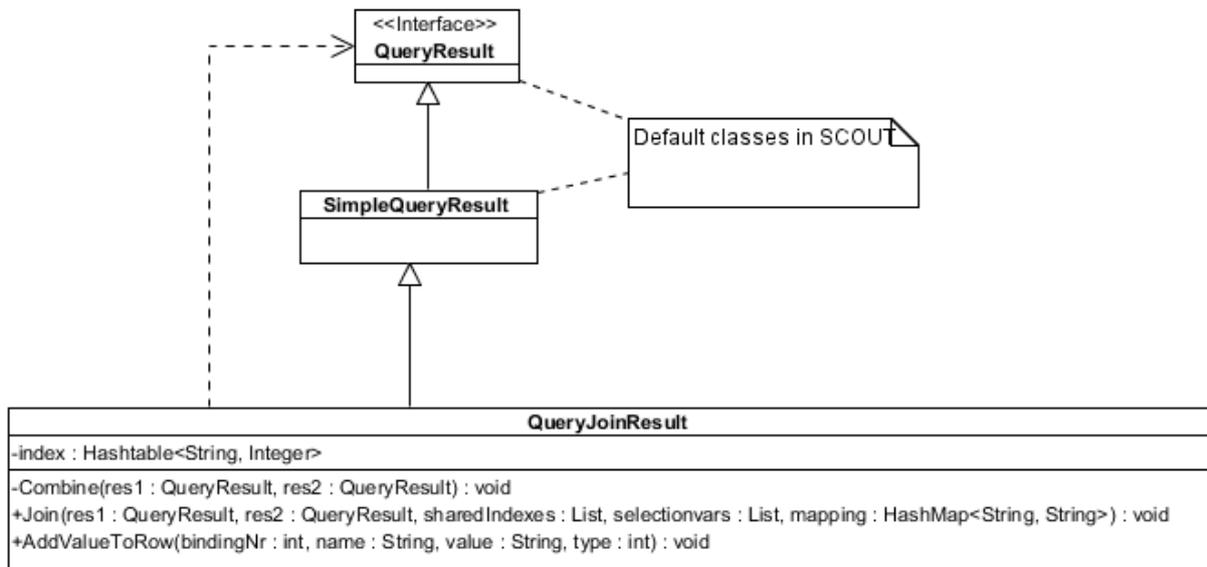


Fig. 16. QueryJoinResult Class Diagram

6.3 Comparing queries

A second step to optimize query execution is the use of a query cache. However, before such a cache can be employed, we must be able to compare the currently executing query to the cached queries in the cache. This section explains how queries can be compared. The first subsection explains how queries are first transformed into a graph. Afterwards, the second subsection explains how two of these graphs are compared.

6.3.1 Building query graphs

While section 5.2 explains the concept behind this query graph building, this section explains how the process is actually implemented. Firstly, one of the methods from the GraphBuilder class will be called, depending on the kind of graph you want to build (i.e. a graph with or without inverse edges). This method instantiates a specific implementation of the Visitor class (from the SPARQL Engine for Java), which then builds the requested type of graph. The following subsections explain each of these implementation steps. First the basic building blocks of the implementation are discussed, and then the implementation of the graph structure is elaborated. Afterwards, the different visitor subclasses are discussed, and finally we elaborate on the GraphBuilder class implementation. Figure 17 shows the class diagram for all of these components.

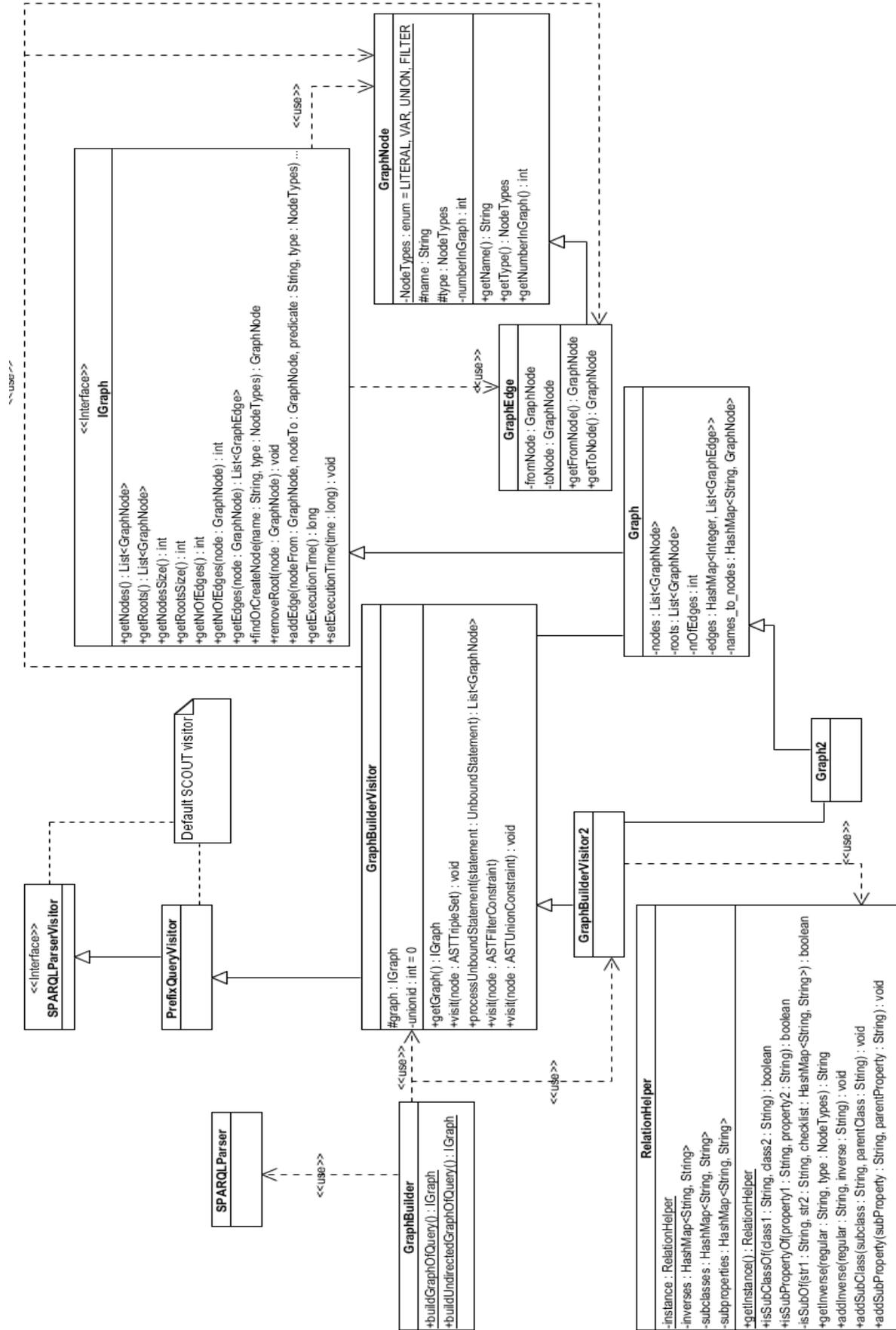


Fig. 17. Graph Class Diagram.

6.3.1.1 Basic building blocks

In this subsection, the two basic building blocks of the graph structure are discussed: the GraphNode class and the GraphEdge class, which respectively represent nodes and edges for use in the graphs.

Since the GraphEdge and GraphNode classes are similar, the GraphEdge class inherits from the GraphNode class, allowing access to all its methods. Therefore, we first discuss the GraphNode class implementation, and then elaborate on the GraphEdge class and its extra methods.

The GraphNode class implements three methods:

- getName: returns the (variable or literal) name of this node
- getType: returns an enum type of LITERAL, VAR, UNION or FILTER, representing the respective node's type.
- getNumberInGraph: returns a unique number used in the graph implementation mentioned in the next subsection.

On top of these three methods inherited from the GraphCode class, the GraphEdge class provides two extra methods:

- getFromNode: returns the source GraphNode of this edge.
- getToNode: returns the destination GraphNode of this edge.

6.3.1.2 Graph structure

This subsection discusses the graph structure used in our implementation. Firstly, we discuss a basic interface, called IGraph, which forms the basis for our two implemented graph structures (together with any future graph structure). Afterwards, the two graph implementations (Graph and Graph2) are discussed.

The IGraph interface has the following methods:

- getNodes: returns a list containing all the GraphNodes available in this graph.
- getRoots: returns a list containing the root GraphNodes available in this graph (if there are any).
- getNodesSize: returns the amount of nodes available in this graph.
- getRootsSize: returns the amount of roots available in this graph.
- getNrOfEdges: returns the total amount of edges available in this graph, if a GraphNode is given, returns the amount of edges starting from this GraphNode.
- GetEdges: returns the edges for a given GraphNode.
- findOrCreateNode: based on a given name and type, this method first tries to find an existing GraphNode which matches these conditions. If none is found, a new GraphNode is created. Afterwards, the found or newly created GraphNode is returned. This method is used by the Visitor implementations mentioned in section 6.3.1.3.
- RemoveRoot: removes the "root status" of a GraphNode. This method is used internally and by the Visitor implementations mentioned in section 6.3.1.3.
- addEdge: creates a GraphEdge between two GraphNodes and stores it in the graph.
- getExecutionTime: returns the time needed to build the graph.

- `setExecutionTime`: sets the time needed to build the graph and is used by the Visitor implementations mentioned in section 6.3.1.3.

The first concrete implementation of this interface is the `Graph` class, which is used to represent a SPARQL query's underlying graph structure. It creates a node for each of the subjects and objects found in the query, and creates edges between them based on the specified predicates. This implementation is based on the Adjacency Lists representation found in [40].

The `Graph` class keeps five fields:

- `nodes`: the list containing all the `GraphNode`s.
- `roots`: the list containing all the roots.
- `nrOfEdges`: the amount of edges that can be found in this graph.
- `edges`: a hashmap containing a list with `GraphEdges` for each of the `GraphNode`s available, using the `GraphNode`s as keys.
- `names_to_nodes`: a hashmap containing a simple mapping from name to `GraphNode`; this method is used by the `findOrCreateNode` method mentioned above.

The second concrete implementation is the `Graph2` class, which further extends the `Graph` class. This class is used to construct a graph which, in addition to the regular edges, contains the inverse edges between the nodes. There is only one difference with the `Graph` class mentioned before; the `addEdge` method also adds the inverse edges between node B and A using the `RelationHelper` class, that currently can be populated manually (future work could provide a way to do this automatically based on one or more ontologies).

The `RelationHelper` class is a singleton class and contains three fields:

- `inverses`: a hashmap containing all the inverse relationships, using the regular relationship as key.
- `subclasses`: a hashmap containing all subclasses using the subclass as key.
- `subproperties`: a hashmap containing all subproperties using the subproperty as key

And has the following methods:

- `isSubClassOf`: checks whether the first class is a subclass of the second
- `isSubPropertyOf`: checks whether the first property is a subproperty of the second
- `isSubOf`: internal helper procedure, which checks if the value associated with a given key in a given hashmap is the same as a given value.
- `getInverse`: returns the inverse of a given relation, or the same if no inverse was found
- `addInverse`: adds the inverse relationship between two relations
- `addSubClass`: adds the relationship between a subclass and its parentclass
- `addSubProperty`: adds the relationship between a subproperty and its parent property

6.3.1.3 Visitor classes

This subsection discusses the Visitor class implementations used to build the graphs mentioned in the previous subsection. For each of the graph classes `Graph` and `Graph2`, a Visitor subclass (more specifically, a `PrefixQueryVisitor` subclass) has been implemented.

The first implementation is the `GraphBuilderVisitor`, which inherits from the `PrefixQueryVisitor` (an already existing `Visitor` implementation in the SCOUT framework) and is used to create instances of the `Graph` class. `GraphBuilderVisitor` implements the following methods:

- `visit(ASTTripleSet)`: detects triple patterns, and adds their subjects and objects as nodes to the graph, and the specified predicate as an edge between these two nodes.
- `visit(ASTFilterConstraint)` : detects filters and adds them as nodes to the graph.
- `visit(ASTUnionConstraint)`: detects UNION clauses and adds them to the graph as a node of type UNION. Afterwards the triples found in these unions are processed and edges between the nodes created by these triples and the UNION node are created.

The `GraphBuilderVisitor` contains the following fields:

- `graph`: a specific `IGraph` implementation.
- `unionid`: used to create unique ids for each union node.

The `GraphBuilderVisitor` also contains the following extra methods:

- `getGraph`: returns the created `IGraph` instance.
- `processUnboundStatement`: a helper method for adding triples to the graph.

The second implementation `GraphBuilderVisitor2` inherits from `GraphBuilderVisitor`, only changing the type of graph returned (i.e. `Graph2` instead of `Graph1`). Note that the `Graph2` implementation itself is responsible for adding the inverse edges to the graph in the `addEdge` method (see previous subsection).

6.3.1.4 GraphBuilder

This subsection discusses the `GraphBuilder` implementation, which contains methods to create a specific `Graph` (subclass) instance corresponding to a given query. It does this by using the `SPARQLParser` found in the `SPARQL Engine for Java` to parse the query into an AST and then uses a specific `GraphBuilderVisitor` implementation to visit the constructed AST and extract the required data from the query.

The `GraphBuilder` implements two methods:

- `buildGraphOfQuery`: which builds a `Graph` instance using `GraphBuilderVisitor`.
- `buildUndirectedGraphOfQuery`: which builds a `Graph2` instance using `GraphBuilderVisitor2`.

6.3.2 Comparing query graphs

This subsection describes the implementation of the graph comparison algorithm, following the rules describes in section 5.3. To allow for other implementations to be added in the future, we first introduced an abstract class called `AbstractGraphCompare`. The concrete implementation of this class is the `GraphCompare` class. `GraphCompare` also makes use of the `RelationHelper` class mentioned before. Figure 18 shows the class diagram for these components.

The `AbstractGraphCompare` implementation consists of the following fields:

- `comp_map`: a comparison map containing the nodes found in the target graph, binding them to the nodes in the source graph.

- source: the source graph used in the comparison.
- target: the target graph used in the comparison.

The class provides the following three methods:

- getComparisonMap: returns the comp_map.
- compare: compares the source and target graph, and returns a number representing the outcome: 0 when the two graphs are equal, -1 when the graphs are not equal, 1 if the source graph is part of the target graph (but is still smaller).
- SameType: compares two nodes (or edges, since GraphEdge is a subclass of GraphNode), and checks whether they are the same, based on type of the node (or edge) and existing sub class and sub property relations. To achieve this, this method contacts the RelationHelper class.

The GraphCompare implementation, which inherits from AbstractGraphCompare, contains the following fields:

- visitedNodes: a stack containing all the nodes visited at a certain time, which prevents loops from occurring, when traversing the graph.
- foundNodes: contains a list of nodes in the target graph, that do not have to be traversed anymore.

Next to these fields, GraphCompare also implements the following methods:

- findPathFromNode: for a given node found in the source graph, this method tries to find a path in the target graph that is similar (e.g. the same or containing subclasses/subproperties).
- compareEdges: compares all the edges between two given nodes.
- Traverse: continues following the current path until either the entire path is found, or a part of the path cannot be found anymore.

See section 5.3.1 for more information.

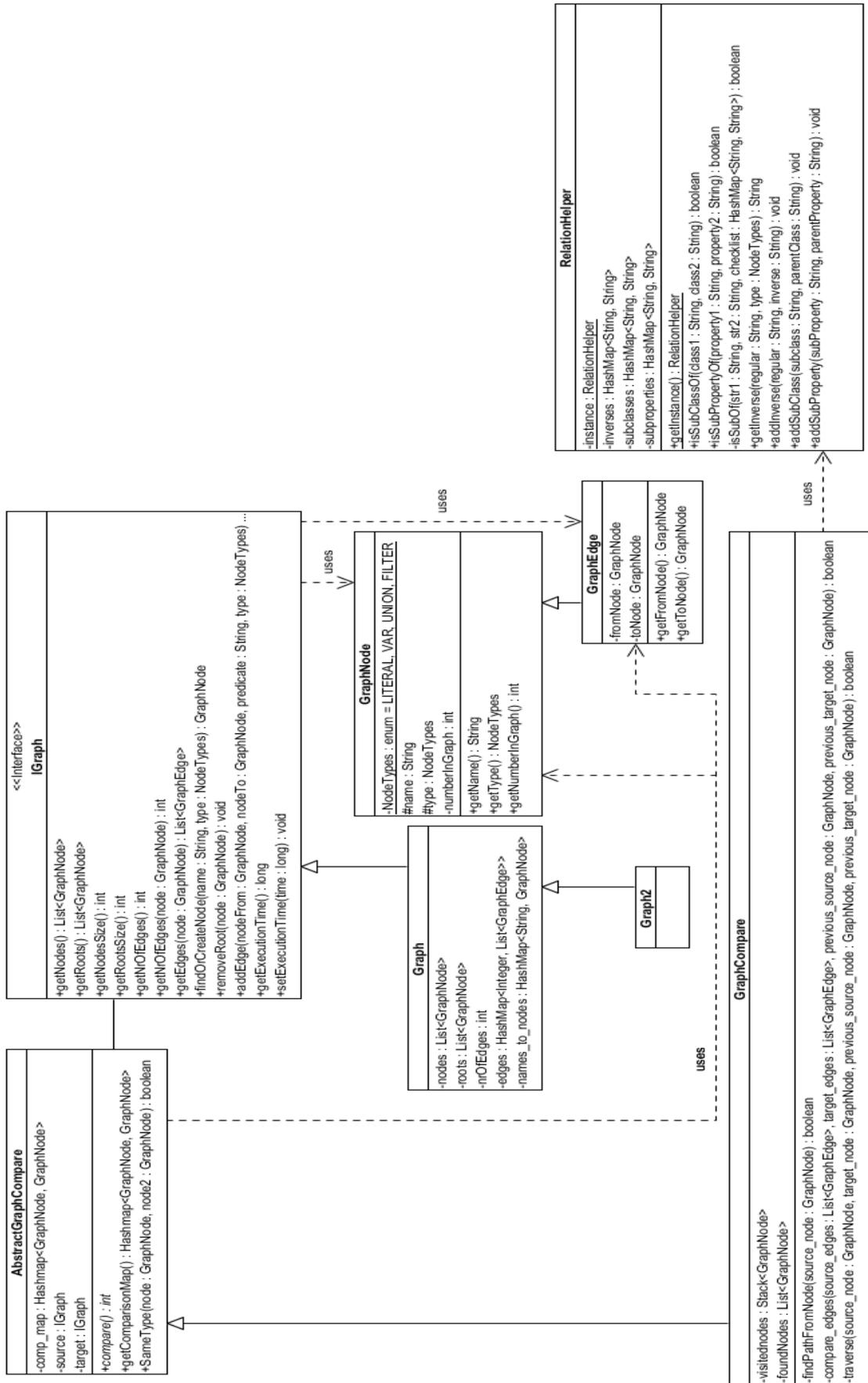


Fig. 18. Graph Compare Class Diagram.

6.4 Caching strategy

This section covers the implementation of the cache component, which realizes the second step in the query optimization process. This component is responsible for storing cache elements and deciding which have to be removed, using a specific eviction strategy, whenever room is required. Below the abstract ICache interface is discussed. The next subsections will describe our implementation of the LRU eviction strategy. Finally, the CacheManager is described, which handles retrieving cached results based on a given query.

ICache is implemented as an interface, which currently only has one implementation: CacheLRU. CacheLRU is a cache implementation that uses Least Recently Used (LRU) as an eviction strategy, and is based on the already existing CacheLRU implementation in the SCOUT framework (used for caching sources). For more information on LRU we refer to section 5.4.

The elements stored in the cache are represented by the abstract CacheElement class. As discussed in section 6.1, for the purpose of this thesis, query results will be stored. Because of this, we mention only one concrete implementation of CacheElement called QueryCacheElement, which is used to store query results. Every implementation of a cache element must provide access to its value and size, which is calculated in bytes. Figure 19 shows the class diagram for these components.

The ICache interface provides ten methods:

- add: adds a cache element to the cache.
- update: updates the cache element for a particular key.
- isUsed: updates the timestamp for a given key.
- contains: checks whether a certain key is available in the cache.
- remove: used to remove cache elements from the cache.
- getKeys: returns a list containing all keys available in the cache.
- getCacheElement: returns the CacheElement instance bound to a specific key.
- getAmountOfHits: the amount of times cache elements were retrieved from the cache.
- getAmountOfEvits: the amount of times cache elements have been removed during the life time of the cache.
- getSize: returns the current amount of bytes used in the cache.

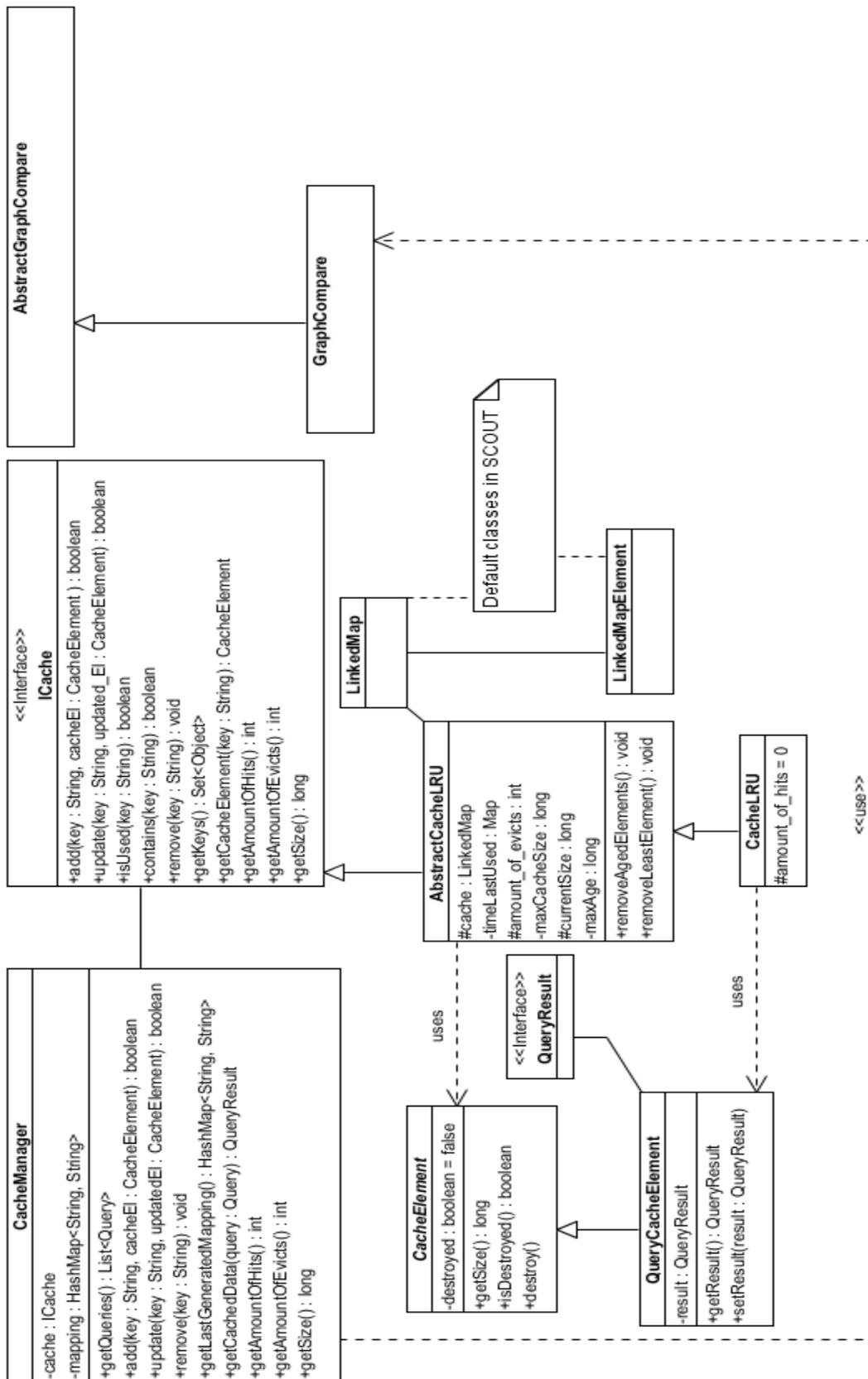


Fig. 19. Cache Class Diagram.

6.4.1 Least Recently Used

This subsection explains the implementation details of CacheLRU as created for the purpose of this thesis. First, the abstract class AbstractCacheLRU is discussed, which is based on the already existing implementation of LRU in SCOUT. This abstract class implements the ICache interface mentioned above. Secondly the specific implementation of this class, called CacheLRU, for use in this thesis is elaborated.

AbstractCacheLRU has six fields:

- Cache: a data structure (LinkedMap, already available in SCOUT), which stores both an ordered and a linked list of elements, so that less recent elements can be found faster.
- timeLastUsed: a HashMap containing the timestamps for each cache element.
- amount_of_evicts: keeps the amount of evictions that occurred in the cache (not present in the original implementation).
- maxCacheSize: keeps the maximum amount of bytes the cache is allowed to contain.
- currentSize: keeps the current amount of bytes the cache contains.
- maxAge: the maximum amount of time a cache element is allowed to remain in the cache in milliseconds.

AbstractCacheLRU provides a basic implementation for the following methods of the ICache interface:

- add
- update
- isUsed
- contains
- remove

AbstractCacheLRU provides the following extra methods:

- getSize(key, CacheElement): returns the size in bytes for a specific cache element (used internally).
- removeLeastElement: removes the element out of the cache that was the most recently used one.
- removeAgedElements: removes all elements out of the cache that have expired timestamps.

CacheLRU adds one additional field:

- amount_of_hits: stores the amount of successful retrievals of cached elements.

CacheLRU also provides an implementation for the following methods from the ICache interface:

- getKeys
- getCacheElement
- getAmountOfHits
- getAmountOfEvicts
- getSize

6.4.2 Cache Manager

This subsection describes the implementation of the Cache manager, which is a layer built on top of the cache strategy mentioned in the previous subsection. The Cache manager is used to retrieve information about queries available in the cache, and to retrieve query results from the cache based on a given query. Since queries might use different variable naming schemes, this class constructs a mapping for these variables (e.g. used to translate variable names from graph B to the variable names used in graph A) that can be used by whichever class is accessing the cache.

The CacheManager has two fields:

- cache: contains the employed caching strategy (in the case of this thesis CacheLRU).
- mapping: a hashmap containing a one on one mapping between the variables found in the last successful query result retrieval from the cache.

The CacheManager provides the following methods:

- getQueries: returns a list of queries that can be found in the cache.
- add: adds a query to the underlying caching layer (i.e. caching strategy).
- update: updates a query in the underlying caching layer.
- remove: removes a query from the underlying caching layer.
- getLastgeneratedMapping: returns the mapping of variables.
- getCachedData: checks whether the given query is available in the underlying cache. If so, this method retrieves the corresponding cache element, after which the query results are returned and a variable mapping is built.
- getAmountOfHits: returns the amount of hits found in the underlying cache layer.
- getAmountOfEvicts: returns the amount of evictions found in the underlying cache layer.
- getSize: returns the current memory size of the underlying cache layer.

Chapter 7. Evaluation

This thesis investigated how the query mechanism, used by the Query and Notification Service, can be optimized. To fulfill this purpose, we have investigated two query pipeline strategies for use by Notification Service queries. Secondly we have investigated a query caching strategy, which can be used to cache the results of the queries posed to the query mechanism.

This chapter is dedicated to the evaluation of the two developed mechanisms. The first section describes the overall test environment. The next two sections present and evaluate the test results of each mechanism.

7.1 Testing environment

The test environment consists of a client part (i.e. the SCOUT framework, extended with the strategies presented in this thesis), which is run on a Samsung Galaxy Apollo (I5800). The device has a 667MHz CPU, 246MB RAM and runs the Android 2.2 operating system.

Every test case is executed 10 times spread over several days (ideally, each test should have been executed more, but due to a lack of time this was not possible). All the results mentioned in this chapter are averaged over these 10 runs. The test cases use sixty-six RDF sources spread over three locations. The distribution of these sources is as follows: 23 sources on www.snakesvx.net, 14 sources on tania.snakesvx.net and 29 sources on wilma.vub.ac.be (www.snakesvx.net and tania.snakesvx.net are deployed on the same server).

7.2 Query Pipeline

This section evaluates the three in the query pipeline process, together with the associated strategies in each step. This evaluation takes into account the following criteria: time needed to execute the original query; time needed to analyze a given query; time needed to construct the sub queries; execution time of the generated sub queries; execution time to join the results generated by the sub queries; time needed to execute the entire test; amount of results for each query using the two strategies (i.e. join strategy vs. injection strategy, see subsection 7.2.8).

Since the analysis and joining of the sub query results is all done using a single strategy (i.e. manual query analysis, see subsection 7.2.2), the main focus is put on the difference between the two sub query construction strategies. In the first construction strategy (join strategy) the sub queries are directly extracted from the original query. In the second construction strategy (injection strategy), the second sub query receives an additional FILTER element, that limits the results requested in the second query, based on the values for the shared variables found in the results of the first sub query. In addition, we consider a worst-case strategy (BasicTest), where the query pipeline is not used and the original query is executed on both the new source as well as the existing sources found in the Environment Model.

7.2.1 Test cases

In order to evaluate the query pipeline, four test cases are used, numbered from 1 – 4. Test case 1 (BasicTest) is used for timing the execution of each query and obtaining the total time required to execute the entire test. Test cases 2 through 4 are used to measure the construction time for each of the construction strategies. Test case 2 is used to determine the time required to analyze a given query. Test cases 3 and 4 are used to measure the time required to execute each sub query, and to

join the sub query results. These test cases are also used to measure the execution time of each query (and the entire test) and the amount of query results returned by each construction strategy.

Each test case performs the same basic steps, allowing relevant data (e.g. query construction time, query execution time) for each mechanism to be extracted. Each test therefore starts by adding 29 sources to the Source Index Model (SIM, see section 2.1.3.2) used in our tests, after which every query is executed once on the query-relevant sources, as determined by the SIM. After having done this warm up phase, 14 new sources are added to the SIM after which each query is executed again. Subsequently, another 8 sources are added, and the queries are executed again. Finally, 15 more sources are added after which queries are executed again. This strategy is used to compare the execution times using each of the queries in different situations, starting from a basic environment which is expanded over time. Adding a different amount of sources every time, allows us to determine the effect these sources have on the overall execution time.

7.2.1.1 Test queries

This subsection describes the “given” queries used in all of the test cases. These queries are used to simulate a series of queries stored in the Notification Service, awaiting execution on a newly discovered source. Note that the complexity of these queries varies from low to medium. This allows us to observe possible differences in execution performance between queries of varying complexity. Each query has two versions; query a is used for the regular tests, while query b is used for tests involving the query pipeline (and thus written using named graphs, since there is a high chance that queries added by a single application will be similar, we decided to create queries which emulate a possible scenario where this occurs).

The first query retrieves all products sold by the shop found in the newly discovered source, which have the same type as any product you have encountered before.

```
PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX sumo: <http://www.ontologyportal.org/SUMO.owl#>

SELECT ?shop ?item
WHERE
{
    ?shop a sumo:RetailStore .
    ?shop region:sells ?item .
    ?item a sumo:Product .
    ?item a ?type .
    ?shop2 a sumo:RetailStore .
    ?shop2 region:sells ?item2 .
    ?item2 a sumo:Product .
    ?item2 a ?type .
}
```

SPARQL Query 14. Test query 1a.

```
PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX sumo: <http://www.ontologyportal.org/SUMO.owl#>

SELECT ?shop ?item
WHERE
{
    GRAPH <http://wise.vub.ac.be/querypart/part1/> {
        ?shop a sumo:RetailStore .
        ?shop region:sells ?item .
        ?item a sumo:Product .
        ?item a ?type .
    }
}
```

```

    GRAPH <http://wise.vub.ac.be/querypart/part2/> {
      ?shop2 a sumo:RetailStore .
      ?shop2 region:sells ?item2 .
      ?item2 a sumo:Product .
      ?item2 a ?type .
    }
  }
}

```

SPARQL Query 15. Test query 1b.

The second query is a variation of the previous one, where the order of the triples in the query has been changed, together with one of the variable names.

```

PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX sumo: <http://www.ontologyportal.org/SUMO.owl#>

SELECT ?s ?item
WHERE
{
  ?shop2 a sumo:RetailStore .
  ?shop2 region:sells ?item2 .
  ?item2 a sumo:Product .
  ?item2 a ?type .
  ?s a sumo:RetailStore .
  ?s region:sells ?item .
  ?item a sumo:Product .
  ?item a ?type .
}

```

SPARQL Query 16. Test query 2a.

```

PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX sumo: <http://www.ontologyportal.org/SUMO.owl#>

SELECT ?s ?item
WHERE
{
  GRAPH <http://wise.vub.ac.be/querypart/part2/> {
    ?shop2 a sumo:RetailStore .
    ?shop2 region:sells ?item2 .
    ?item2 a sumo:Product .
    ?item2 a ?type .
  }
  GRAPH <http://wise.vub.ac.be/querypart/part1/> {
    ?s a sumo:RetailStore .
    ?s region:sells ?item .
    ?item a sumo:Product .
    ?item a ?type .
  }
}

```

SPARQL Query 17. Test query 2b.

The third query is a bit more specific than the first two. This query retrieves encountered items with the same type as the beds sold at the newly encountered store, together with the shop selling those encountered items.

```

PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX sumo: <http://www.ontologyportal.org/SUMO.owl#>

SELECT ?shop2 ?item2
WHERE
{
  ?item a sumo:Bed .
  ?item dc:title ?title .
  ?item a ?type .
  ?item2 a sumo:Product .
  ?item2 a ?type .
}

```

```

?shop2 a sumo:RetailStore .
?shop2 region:sells ?item2 .
}

```

SPARQL Query 18. Test query 3a.

```

PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX sumo: <http://www.ontologyportal.org/SUMO.owl#>

SELECT ?shop2 ?item2
WHERE
{
  GRAPH <http://wise.vub.ac.be/querypart/part1/> {
    ?item a sumo:Bed .
    ?item dc:title ?title .
    ?item a ?type .
  }
  GRAPH <http://wise.vub.ac.be/querypart/part2/> {
    ?item2 a sumo:Product .
    ?item2 a ?type .
    ?shop2 a sumo:RetailStore .
    ?shop2 region:sells ?item2 .
  }
}

```

SPARQL Query 19. Test query 3b.

The fourth query, another variation of the first query, retrieves the name of the encountered shop together with the name of the sold products, which are related to products of the same type encountered in the past. The name of the shops and the products encountered in the past are also requested.

```

PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX sumo: <http://www.ontologyportal.org/SUMO.owl#>

SELECT ?shoptitle ?title ?shop2 ?item2
WHERE
{
  ?shop a sumo:RetailStore .
  ?shop region:sells ?item .
  ?shop dc:title ?shoptitle .
  ?item a sumo:Product .
  ?item dc:title ?title .
  ?item a ?type .
  ?shop2 a sumo:RetailStore .
  ?shop2 region:sells ?item2 .
  ?item2 a sumo:Product .
  ?item2 a ?type .
}

```

SPARQL Query 20. Test query 4a.

```

PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX sumo: <http://www.ontologyportal.org/SUMO.owl#>

SELECT ?shoptitle ?title ?shop2 ?item2
WHERE
{
  GRAPH <http://wise.vub.ac.be/querypart/part1/> {
    ?shop a sumo:RetailStore .
    ?shop region:sells ?item .
    ?shop dc:title ?shoptitle .
    ?item a sumo:Product .
    ?item dc:title ?title .
  }
}

```

```

        ?item a ?type .
    }
    GRAPH <http://wise.vub.ac.be/querypart/part2/> {
        ?shop2 a sumo:RetailStore .
        ?shop2 region:sells ?item2 .
        ?item2 a sumo:Product .
        ?item2 a ?type .
    }
}

```

SPARQL Query 21. Test query 4b.

The fifth retrieves the restaurant and the cuisine found in the new source, where the cuisine is related to a cuisine encountered in previous restaurants.

```

PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX rest: <http://gaia.fdi.ucm.es/ontologies/restaurant.owl#>

SELECT ?restaurant ?cuisine
WHERE
{
    ?restaurant a region:Restaurant .
    ?restaurant rest:typeOfCuisine ?cuisine .
    ?cuisine a ?type .
    ?restaurant2 a region:Restaurant .
    ?restaurant2 rest:typeOfCuisine ?cuisine2 .
    ?cuisine2 a ?type .
}

```

SPARQL Query 22. Test query 5a.

```

PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX rest: <http://gaia.fdi.ucm.es/ontologies/restaurant.owl#>

SELECT ?restaurant ?cuisine
WHERE
{
    GRAPH <http://wise.vub.ac.be/querypart/part1/> {
        ?restaurant a region:Restaurant .
        ?restaurant rest:typeOfCuisine ?cuisine .
        ?cuisine a ?type .
    }
    GRAPH <http://wise.vub.ac.be/querypart/part2/> {
        ?restaurant2 a region:Restaurant .
        ?restaurant2 rest:typeOfCuisine ?cuisine2 .
        ?cuisine2 a ?type .
    }
}

```

SPARQL Query 23. Test query 5b.

The sixth query retrieves previously encountered persons, who have the same gender and at least one interest in common with the person currently encountered.

```

PREFIX foaf: <http://xmlns.com/foaf/spec/>

SELECT ?person2 ?gender ?topic
WHERE
{
    ?person a foaf:Person .
    ?person foaf:gender ?gender .
    ?person foaf:interest ?interest .
    ?interest foaf:topic ?topic .
    ?person2 a foaf:Person .
    ?person2 foaf:gender ?gender .
    ?person2 foaf:interest ?interest2 .
    ?interest2 foaf:topic ?topic .
}

```

SPARQL Query 24. Test query 6a.

```

PREFIX foaf: <http://xmlns.com/foaf/spec/>

SELECT ?person2 ?gender ?topic
WHERE
{
    GRAPH <http://wise.vub.ac.be/querypart/part1/> {
        ?person a foaf:Person .
        ?person foaf:gender ?gender .
        ?person foaf:interest ?interest .
        ?interest foaf:topic ?topic .
    }
    GRAPH <http://wise.vub.ac.be/querypart/part2/> {
        ?person2 a foaf:Person .
        ?person2 foaf:gender ?gender .
        ?person2 foaf:interest ?interest2 .
        ?interest2 foaf:topic ?topic .
    }
}

```

SPARQL Query 25. Test query 6b.

The seventh query is a variation of the previous one, with the difference that the persons being compared do not need to have the same gender. Apart from this, the query also uses different names for some of its variables.

```

PREFIX foaf: <http://xmlns.com/foaf/spec/>

SELECT ?p2 ?gdr ?tpc
WHERE
{
    ?person a foaf:Person .
    ?person foaf:interest ?interest .
    ?interest foaf:topic ?tpc .
    ?p2 a foaf:Person .
    ?p2 foaf:gender ?gdr .
    ?p2 foaf:interest ?interest2 .
    ?interest2 foaf:topic ?tpc .
}

```

SPARQL Query 26. Test query 7a.

```

PREFIX foaf: <http://xmlns.com/foaf/spec/>

SELECT ?p2 ?gdr ?tpc
WHERE
{
    GRAPH <http://wise.vub.ac.be/querypart/part1/> {
        ?person a foaf:Person .
        ?person foaf:interest ?interest .
        ?interest foaf:topic ?tpc .
    }
    GRAPH <http://wise.vub.ac.be/querypart/part2/> {
        ?p2 a foaf:Person .
        ?p2 foaf:gender ?gdr .
        ?p2 foaf:interest ?interest2 .
        ?interest2 foaf:topic ?tpc .
    }
}

```

SPARQL Query 27. Test query 7b.

The eighth query is another variation of test query 6. This time only one constraint is removed, namely that persons need to share atleast one interest.

```
PREFIX foaf: <http://xmlns.com/foaf/spec/>

SELECT ?person2 ?gender ?topic
WHERE
{
    ?person a foaf:Person .
    ?person foaf:gender ?gender .
    ?person2 a foaf:Person .
    ?person2 foaf:gender ?gender .
    ?person2 foaf:interest ?interest2 .
    ?interest2 foaf:topic ?topic .
}
```

SPARQL Query 28. Test query 8a.

```
PREFIX foaf: <http://xmlns.com/foaf/spec/>

SELECT ?person2 ?gender ?topic
WHERE
{
    GRAPH <http://wise.vub.ac.be/querypart/part1/> {
        ?person a foaf:Person .
        ?person foaf:gender ?gender .
    }
    GRAPH <http://wise.vub.ac.be/querypart/part2/> {
        ?person2 a foaf:Person .
        ?person2 foaf:gender ?gender .
        ?person2 foaf:interest ?interest2 .
        ?interest2 foaf:topic ?topic .
    }
}
```

SPARQL Query 29. Test query 8b.

The ninth query finds out whether the current resource contains a Sports Accommodation. If so, retrieves the data about all the Sports Accommodations encountered before, that have at least one sport in common.

```
PREFIX region: <http://wise.vub.ac.be/region/>

SELECT ?sports_acc ?sports_acc_2 ?sport
WHERE
{
    ?sports_acc a region:SportsAccommodation .
    ?sports_acc region:accommodatesSport ?sport .
    ?sports_acc_2 a region:SportsAccommodation .
    ?sports_acc_2 region:accommodatesSport ?sport .
}
```

SPARQL Query 30. Test query 9a.

```
PREFIX region: <http://wise.vub.ac.be/region/>

SELECT ?sports_acc ?sports_acc_2 ?sport
WHERE
{
    GRAPH <http://wise.vub.ac.be/querypart/part1/> {
        ?sports_acc a region:SportsAccommodation .
        ?sports_acc region:accommodatesSport ?sport .
    }
    GRAPH <http://wise.vub.ac.be/querypart/part2/> {
        ?sports_acc_2 a region:SportsAccommodation .
        ?sports_acc_2 region:accommodatesSport ?sport .
    }
}
```

SPARQL Query 31. Test query 9b.

The next query retrieves all previously encountered Points of Interest, together with information about the currently encountered Statue.

```
PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?location1 ?location2
WHERE
{
    ?location1 a region:Statue .
    ?location1 dc:title ?title .
    ?location2 a region:PointOfInterest .
    ?location2 dc:title ?title2 .
}
```

SPARQL Query 32. Test query 10a.

```
PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?location1 ?location2
WHERE
{
    GRAPH <http://wise.vub.ac.be/querypart/part1/> {
        ?location1 a region:Statue .
        ?location1 dc:title ?title .
    }
    GRAPH <http://wise.vub.ac.be/querypart/part2/> {
        ?location2 a region:PointOfInterest .
        ?location2 dc:title ?title2 .
    }
}
```

SPARQL Query 33. Test query 10b.

The final query is a variation of the previous one, where the query retrieves information about previously encountered Points of Interest, together with information about the currently encountered Area.

```
PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?location1 ?location2
WHERE
{
    ?location1 a region:Area .
    ?location1 rdfs:label ?label .
    ?location2 a region:PointOfInterest .
    ?location2 dc:title ?title2 .
}
```

SPARQL Query 34. Test query 11a.

```
PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?location1 ?location2
WHERE
{
    GRAPH <http://wise.vub.ac.be/querypart/part1/> {
        ?location1 a region:Area .
        ?location1 rdfs:label ?label .
    }
}
```

```

}
  GRAPH <http://wise.vub.ac.be/querypart/part2/> {
    ?location2 a region:PointOfInterest .
    ?location2 dc:title ?title2 .
  }
}

```

SPARQL Query 35. Test query 11b.

7.2.2 Criterion 1: Analysis time

This section measures the time needed to extract the two different parts from each of the queries mentioned before. Chart 1 shows the test results for the eleven queries. The horizontal axis represents the query number, while the vertical axis represents the time necessary to extract the different parts in milliseconds.

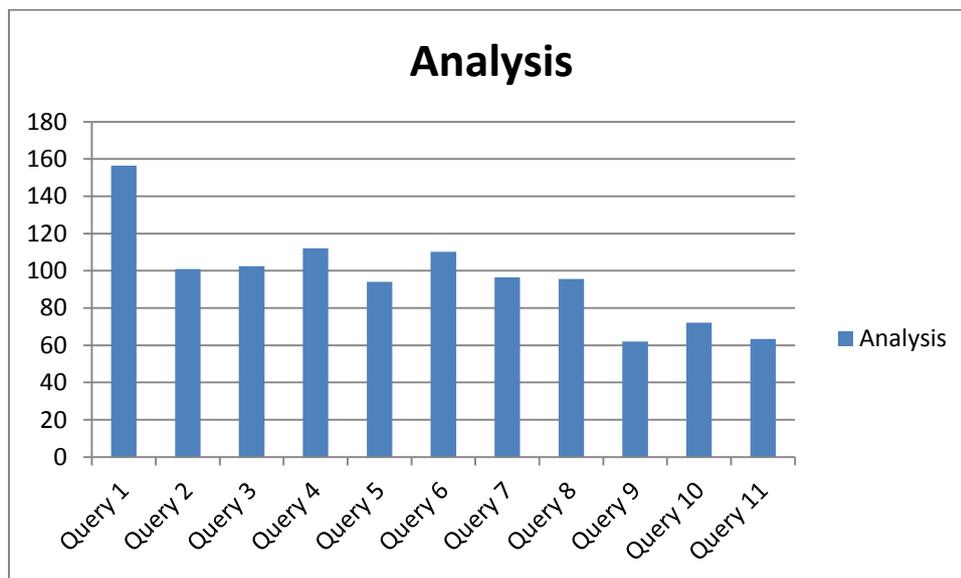


Chart 1. Querypart analys time

We can see that depending on the size of the query the time required to analyze the query increases. This is expected since there are more nodes (in our case triple patterns) that have to be checked and added to the correct query part. Since the times are very small compared to the costs of some of the other criteria (see next subsections), we can conclude that the time required to extract the relevant query information has a negligible impact on the entire query process.

7.2.3 Criterion 2: Construction time

This section measures the time needed to construct the various sub queries for each of the queries. For the second sub query, the necessary time is compared when using the JoinStrategy and the InjectionStrategy. Since the construction of the first sub query occurs the same way in both strategies, the necessary construction time is the same. Chart 2 shows the test results for the eleven queries. The horizontal axis represents the query number, while the vertical axis represents the time necessary to construct the relevant sub queries in milliseconds.

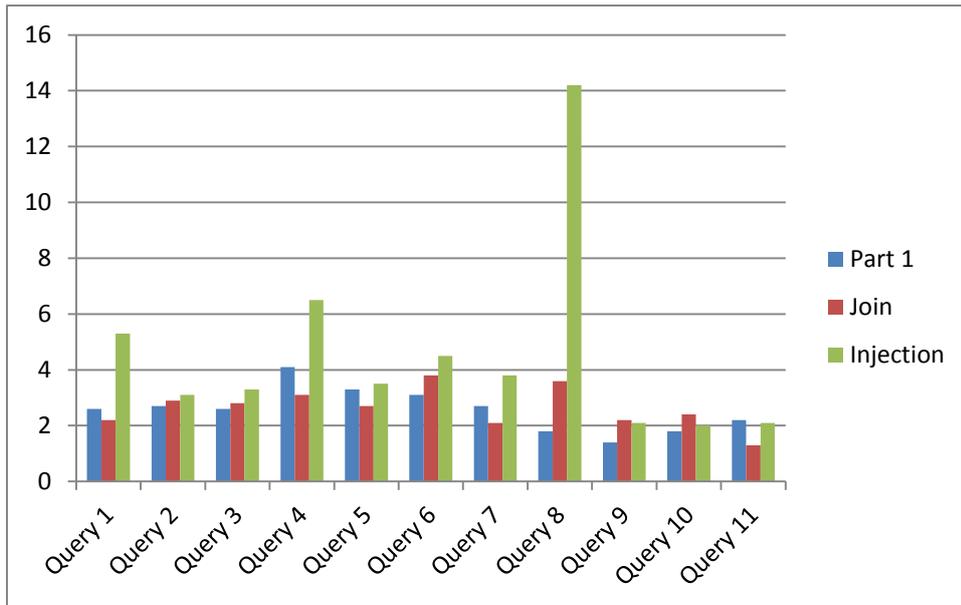


Chart 2. Querypart construction time

As expected, the time required to generate the second sub query takes longer to generate (or the difference is negligible) when using the Injection strategy. This is normal, since both strategies use the same basic mechanism, but the Injection strategy has to generate an extra filter based on results from the first sub query.

Again, when comparing this constructing time to general query execution time, the time is negligible and thus has no effect on the general execution of the query. Note: the peak noticed for query 8, was caused by a single run, in which the construction time took about forty times longer (122ms compared to 3ms).

7.2.4 Criterion 3: Sub query execution time

This section compares the time required to perform each of the sub queries. For the second sub query, the necessary time is compared when using the JoinStrategy and the InjectionStrategy. Since the first sub query is the same in both strategies, only the second sub query will be compared. Chart 2 shows the test results for the eleven queries. The horizontal axis represents the query number, while the vertical axis represents the time necessary to execute the relevant sub queries in milliseconds.

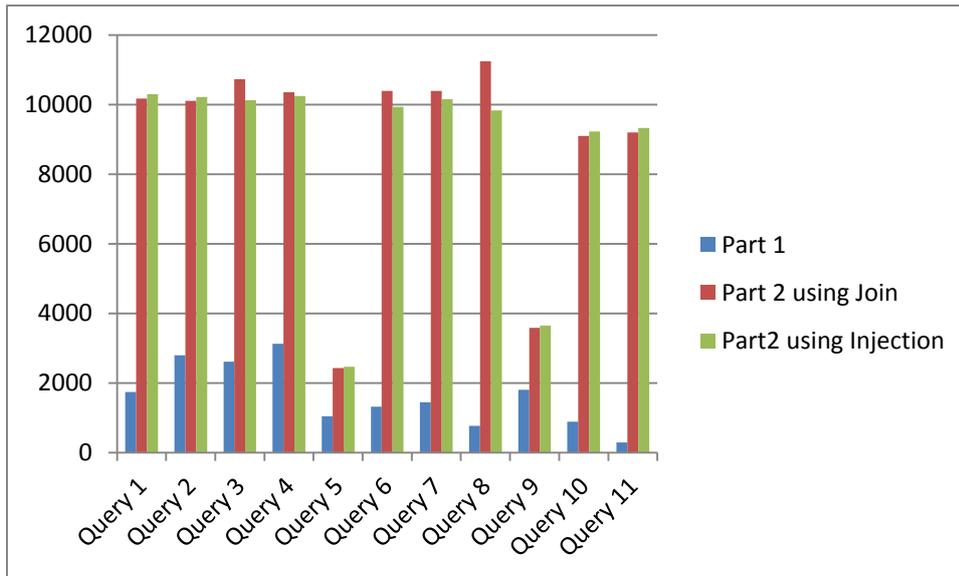


Chart 3. Sub query execution time

As expected, the time required to execute the second sub query takes less time (or the difference is negligible) when using the Injection strategy. This is normal, since both strategies use the same basic mechanism, but the Injection strategy is able to be more restrictive due to the added filter.

7.2.5 Criterion 4: Join time

This section compares the time required to join results for each query based for each of the query pipeline strategies. This test is performed for each of the queries mentioned before. Chart 4 shows the results for each test with on the horizontal axis the query number and on the vertical axis the time required to join the results in milliseconds.

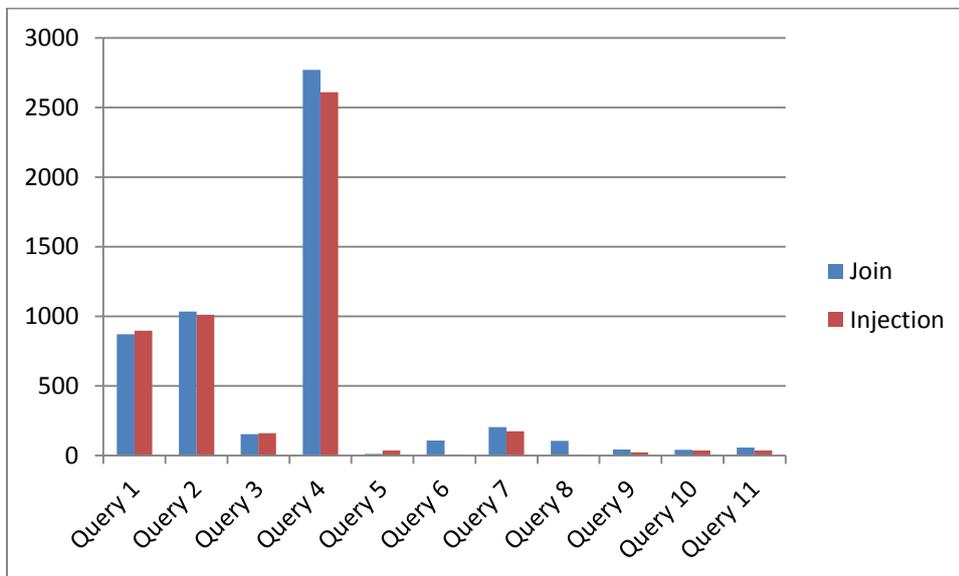


Chart 4. Sub query join time

As expected, since the Injection strategy might return less results, in most cases the time required to join the results using the Injection strategy is either a little faster or about the same as when using the Join strategy. When comparing the data, the time difference measured is only a small

percentage, so that using either strategy is useful. The reason for this time difference is that the second sub query, when using the join strategy, returns more results. This causes the time to combine these results to be higher since more results have to be compared (or copied).

7.2.6 Criterion 5: Execution time of the original query compared to the execution using the query pipeline

This section compares the execution time of the original query on various new sources together with the Environment Model. This test is performed for each of the queries mentioned before and compares the execution time of the original system, to the two query pipeline strategies created in this thesis. Chart 5 shows the results for each test with on the horizontal axis the query number, and on the vertical axis the time required to execute each query in milliseconds.

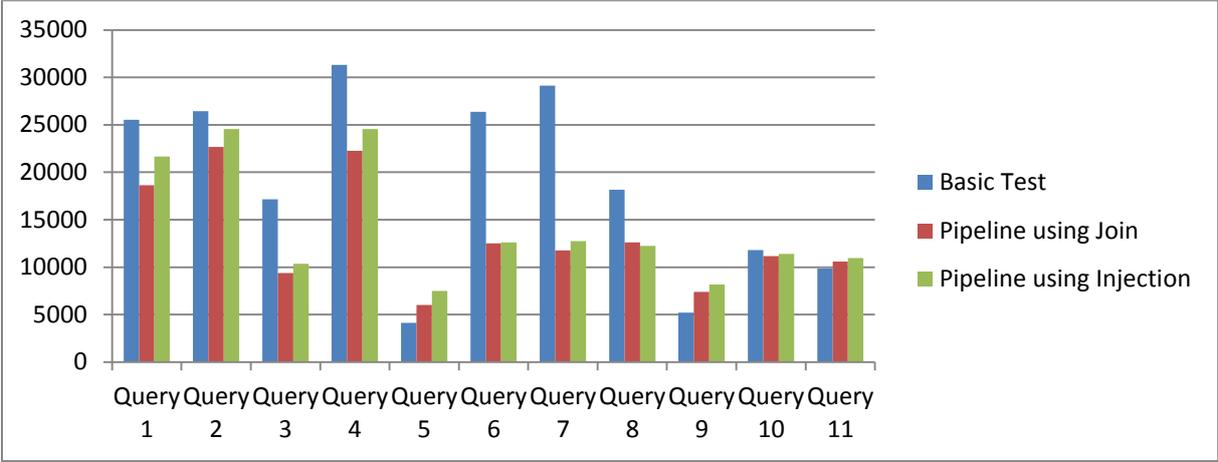


Chart 5. Execution time of original query (compared to pipeline)

When comparing the results of the BasicTest with the two pipeline strategies, we can conclude that for almost all queries, the time required to execute the two strategies is smaller than the standard provided by this test. There are some exceptions though (namely query 5 and 9), in cases where there are not a lot of previously encountered sources, the two pipeline strategies perform a little worse, which is probably caused by the overhead (although very little) during analyses of the original query. This might have an effect in environments where the amount of encountered sources is low (e.g. when starting to use the SCOUT framework), but over time when more and more sources are encountered, the query pipeline can cause a significant performance boost for most queries.

7.2.7 Criterion 6: Execution time of the entire test

This section compares the total execution time of the various tests. More specifically, we compare the time required to execute queries with a query pipeline (using one of the construction strategies), and without (BasicTest). Chart 6 shows the results for these three tests with on the vertical axis the execution time in milliseconds, and on the right which test is represented using which color.

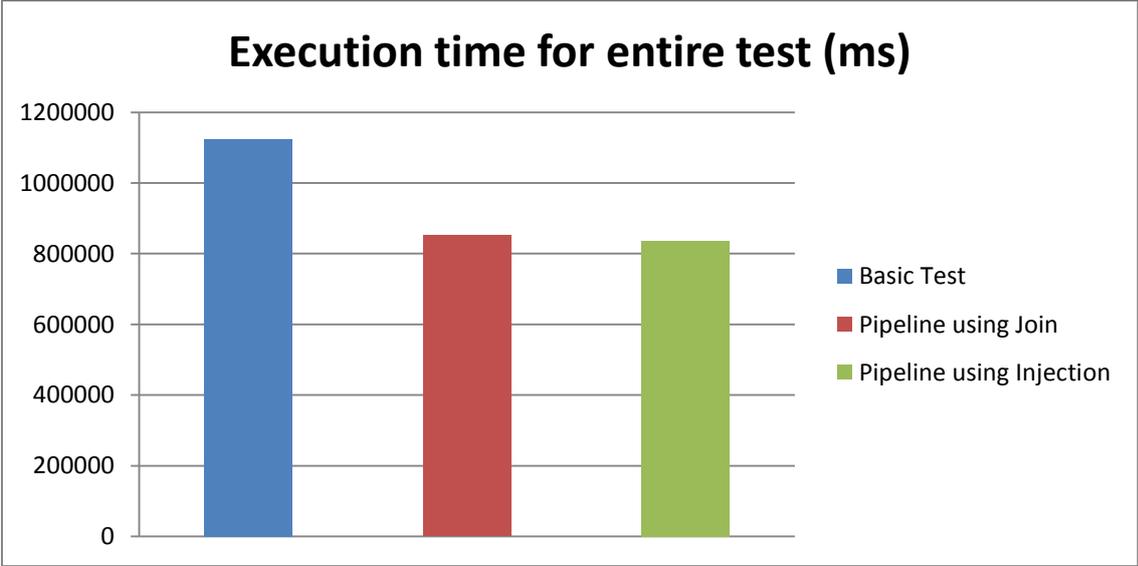


Chart 6. Execution time of each test

From the previous section, we know that in general the execution time of each query is shorter (or has a negligible difference) when using the two pipeline strategies compared to the BasicTest. Between the two strategies, we know that the difference in execution time is very little.

Because of these, it is logical that the execution time for the two pipeline strategy test is closely related to each other, while both of these have a much smaller execution time than the BasicTest (851532ms compared to 1124270ms).

7.2.8 Criterion 7: Amount of results for each query using the two strategies

This section compares the amount of results returned by the two pipeline strategies. This is done to determine whether there is any difference between the two strategies. Chart 7 shows the results of these two tests, on the horizontal axis the query number is shown, while on the vertical axis the amount of results for each query is shown.

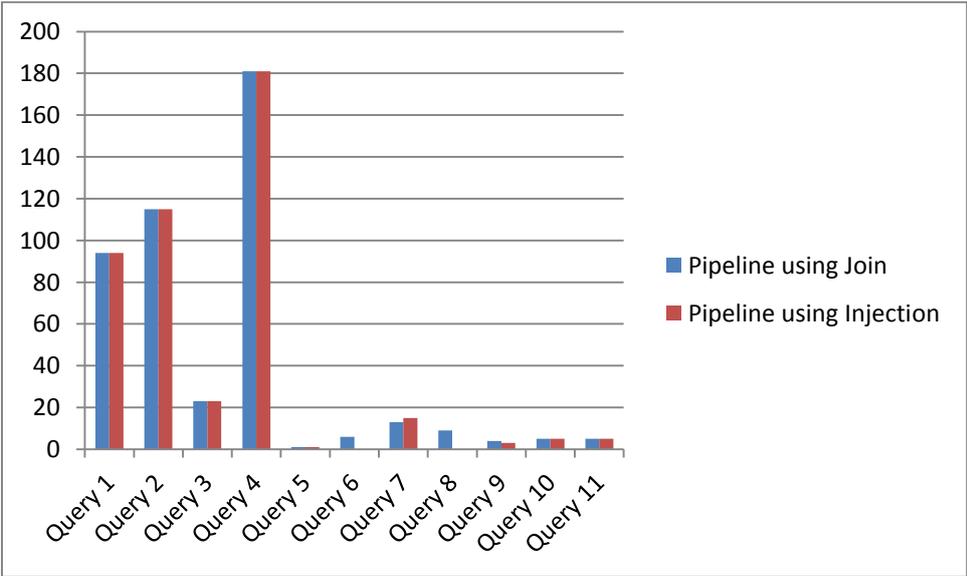


Chart 7. Amount of results using each pipeline strategy

Since both pipeline strategies are fairly similar to each other, we expected that these two strategies would return (almost) the same amount of results for each query after having combined the two sub query results.

When comparing the results, we can notice that in most cases the amount of results returned by both strategies are either the same or close to each other. For some queries the JoinStrategy returned some more results, while for others the InjectionStrategy returned some more. From this we can conclude that there is a small margin of error that has to be taken into account when using both strategies. Since the chart above doesn't show all of the results clearly, below a table is shown with of the concrete figures.

<i>Query Results (phase 1)</i>	<i>Pipeline using Join</i>	<i>Pipeline using Injection</i>
<i>Query 1</i>	<i>94</i>	<i>94</i>
<i>Query 2</i>	<i>115</i>	<i>115</i>
<i>Query 3</i>	<i>23</i>	<i>23</i>
<i>Query 4</i>	<i>181</i>	<i>181</i>
<i>Query 5</i>	<i>1</i>	<i>1</i>
<i>Query 6</i>	<i>6</i>	<i>0</i>
<i>Query 7</i>	<i>13</i>	<i>15</i>
<i>Query 8</i>	<i>9</i>	<i>0</i>
<i>Query 9</i>	<i>4</i>	<i>3</i>
<i>Query 10</i>	<i>5</i>	<i>5</i>
<i>Query 11</i>	<i>5</i>	<i>5</i>

7.2.9 Conclusion

From the first two (test) sections, we have learned that the time required to analyze the given query and constructing the two sub queries has almost no impact on the overall time required to execute a given query. From (test) section 5 and 6, we have learned that the time required for execution using both query pipeline strategies is smaller in most cases, except in cases where the execution time of the given query was already fairly small. From (test) section 7, we learned that there is small margin of error in the amount of results returned by using both strategies.

7.3 Cache

This section evaluates the two steps from the realized query caching mechanism. This evaluation takes into account the following criteria: query graph construction time; query graph comparison time; execution time of the original query; execution time of the total query set; execution time for adding new sources;

7.3.1 Test cases

In order to evaluate the query caching mechanism, especially in relation with the query pipeline mentioned in the previous section, six test cases are used, numbered 1 - 6. The first three test cases are used to measure the execution time for each of the queries, the entire test (and the time needed to add sources to the SIM), the other three focus on caching behavior in particular. Test case 7 is used test graph construction and comparison time. Test cases 5 and 6 are used to test all of the aforementioned criteria, except graph construction and comparison times.

Test case 7 will create a graph for each query and then compare this query to all queries. Test cases 1 through 5 perform the same basic steps, allowing relevant data for each mechanism to be extracted. Each test therefore starts by adding 29 sources to the Source Index Model used in our tests, after which every query is executed once on some of the default sources. After having done this warm up phase, 14 new sources are added to the SIM after which each query is executed again. After this, another 8 sources are added, and the queries are executed again. Finally, 15 more sources are added after which queries get executed again.

7.3.1.1 Test queries

The same queries as mentioned in section 8.2.1.1 are used for these tests.

7.3.2 Criterion 1: Construction time

This section determines how much time is required to construct the graph and to determine the impact this construction has on the overall execution. Chart 8 shows the results for this test, on the horizontal axis the query number is shown, while on the vertical axis the time required for each of the queries is shown in milliseconds.

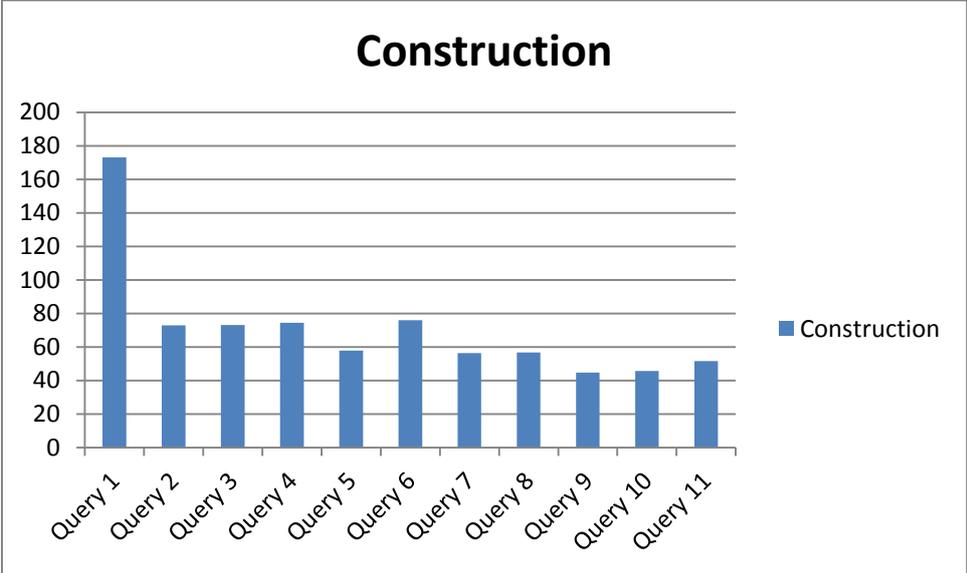


Chart 8. Time needed to construct a query graph

As seen in the chart, the time required in general is very little and will thus not affect performance in a noticeably way. There is a spike in the chart for the first query, but this is caused by a spike during a single run which would have balanced out if more than ten runs might have been done.

7.3.3 Criterion 2: Comparison time

This section is used to determine how much time is required to compare two graphs and to determine the impact this comparison might have on the overall execution. Chart 9 shows the results for this test, on the horizontal axis the query number is shown, while on the vertical axis the time required to compare each of the queries to each other is shown in milliseconds.

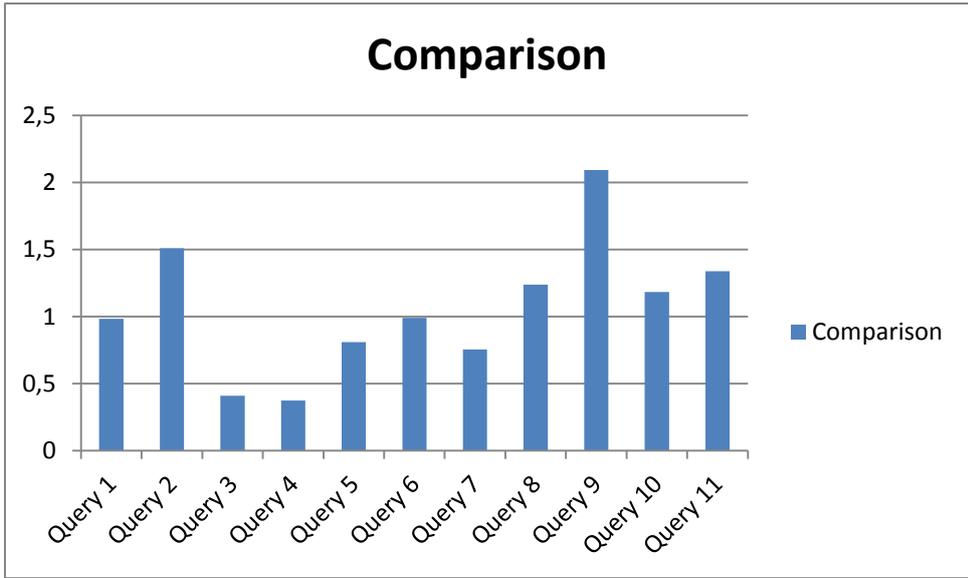


Chart 9. Time needed to compare query graphs

As seen in this chart, the time required to compare two graphs has no noticeable impact on the overall query execution. Even when taking into account the spike noticed for query 9, there is still no noticeable impact.

7.3.4 Criterion 3: Execution time of the original query

This section compares the execution time of the original query on various new sources, together with the Environment Model. This test is performed for each of the queries mentioned before and compares the execution time of the original system, the two query pipeline strategies, and the query caching using the two query pipeline strategies. Chart 10 shows the results for each test with on the horizontal axis the query number, and on the vertical axis the time required to execute each query in milliseconds.

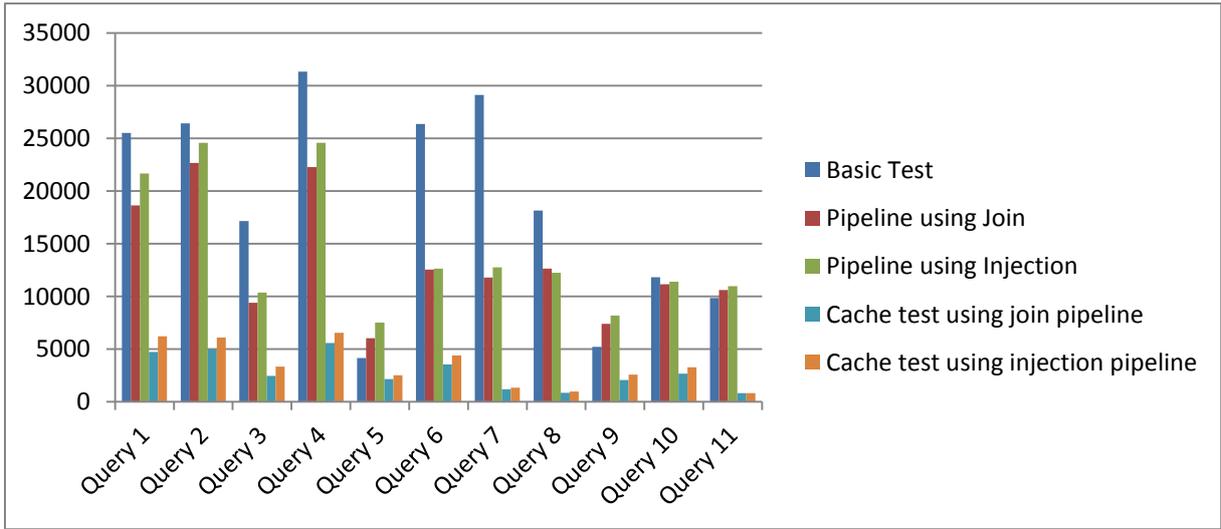


Chart 10. Execution time of the original query using the different strategies.

From section 8.2.6 we already knew that the two query pipeline strategies are better in performance than the BasicTest. When adding the query caching mechanism we can notice that the query execution for each of these tests is even shorter.

When comparing the two query pipeline strategies without vs. with caching, we can notice that the results are improved. Since we know that the query pipeline will always execute the first query on the new source, we can see why caching has a smaller impact on performance.

7.3.5 Criterion 4: Execution time of the entire test

This section compares the total execution time of the various tests. More specifically, we compare the time required to execute queries with a query pipeline, comparing both construction strategies without vs. with caching, and without query pipeline (BasicTest). Chart 11 shows the results for these six tests with on the vertical axis the execution time in milliseconds, and on the right which test is represented using which color.

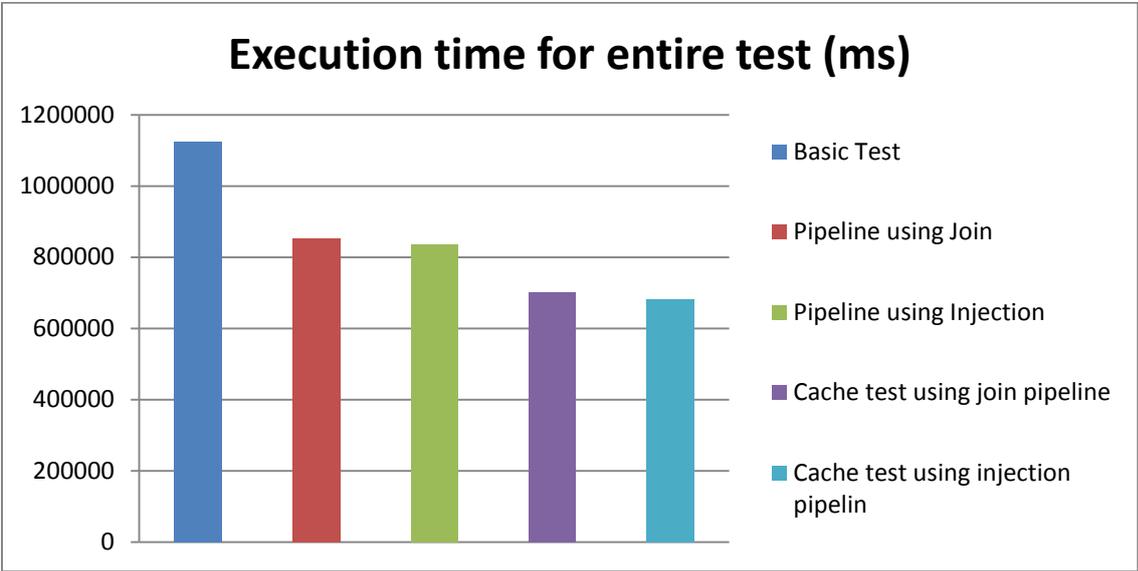


Chart 11. Execution time of each test

From the previous section, we know that in general the execution time of each query is shorter or similar when using the two pipeline strategies compared to the BasicTest and that it is even shorter when using query caching with the query pipeline. Between the two pipeline strategies we know that the difference in execution time is very little.

The difference with the previous section is that the total execution time is considered here (also including updating the cache), and that overall, the difference between the query pipeline without vs. with caching is very little. The reason for this, which is shown in the next section, lies with the updating of the cache whenever a new source is detected.

7.3.6 Criterion 5: Amount of time needed to add new sources

This section compares the total amount of time required to add new sources (and update the cache contents) for each of the tests. Since the BasicTest and the two query pipeline strategies do not use query caching, these are mentioned as a baseline for comparison (only showing the time required to update the SIM). Chart 12 shows the results for each of the tests with on the vertical axis the

execution time in milliseconds, and on the horizontal axis which group of sources was added and on the right, which test is represented using which color.

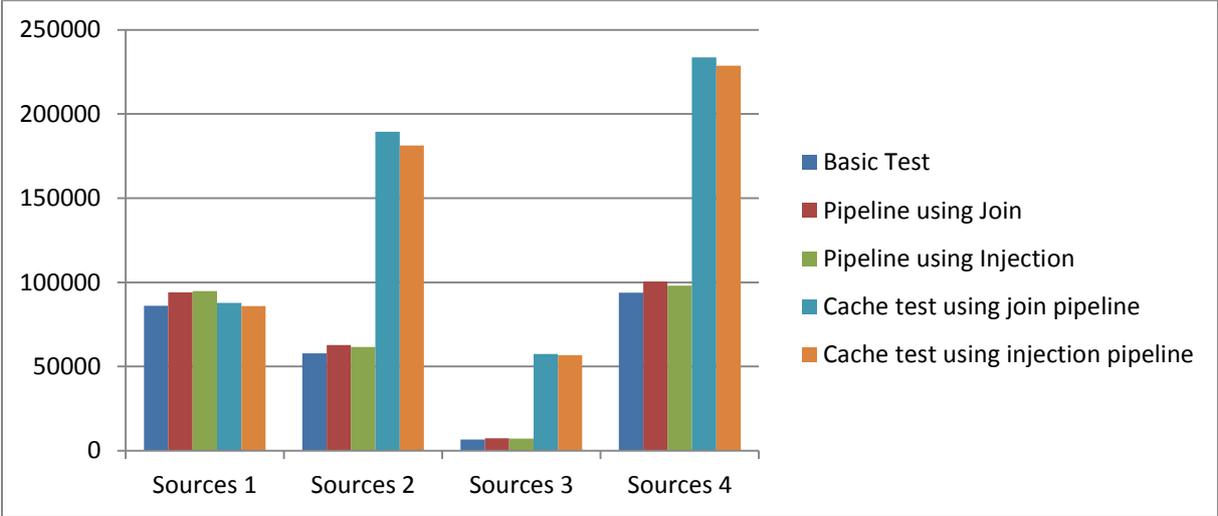


Chart 12. Time needed to add sources (and update the cache).

In the previous section we noticed that the query pipeline together with query caching had the best overall performance, even though the difference with the non-caching pipeline was less than expected. This is because every time a new source is added the cache (i.e. all queries which can retrieve relevant data from this source) has to be updated, reducing the performance gained by using query caching. From this we can learn that in situations where lots of new sources are encountered and not a lot of queries are executed, the caching mechanism can actually use up more resources than needed. A possible solution for this is that for every client, a different timing can be used as lifetime for the cache elements. This is something that can be investigated further in future work.

7.3.7 Conclusion

From the previous section, we already knew that the query pipeline increased performance in most cases. When adding query caching to this, we can notice a smaller increase in performance. Reasons for this are the very small overhead required to compare graphs in the cache to find a possible suitable match and the updating of the cache that happens in the background whenever a new source is encountered. If we would apply this query caching mechanism to the original system, the performance is actually worse, since the cache has to be updated every time a new source is encountered, defeating the use of the cache.

Chapter 8. Conclusion

This thesis has shown how a query pipeline, together with a query caching mechanism, can be used to reduce the time required to execute a query for use with the Notification Server. Experiments were conducted, as explained in chapter 7, to validate these two mechanisms, and to compare several variants with each other. This chapter summarizes the benefits and drawbacks of the developed mechanisms, together with future work that can be done in this area.

8.1 Summary

8.1.1 Query Pipeline

The first mechanism responsible for optimizing the query process allows us to construct sub queries which can be executed separately. This enables us to determine whether or not the second sub query should still be executed based on the results of the first sub query. Originally a strategy was created for analyzing queries and extracting the relevant sub queries, which was able to be used on any query. After some problems occurred regarding the ordering of triples, a new strategy was implemented using named graphs, which was able to tackle these problems. Two query pipeline mechanisms were discussed, which differ in the way the second sub query is constructed.

The first mechanism reuses the parts found in the original query for constructing the second sub query. Therefore, its results can be more effectively cached.

The second mechanism also reuses the parts found in the original query, but adds a filter to the second sub query based on the results of the first. This makes it that these results are more specific and thus takes less time to join together and form the actual results.

Our experimental validation shows us that the second pipeline is preferable to the first strategy when no caching for the second sub query is used.

8.1.2 Cache strategy

The second mechanism responsible for optimizing the query process is the use of query caching. This mechanism keeps query results locally, to avoid having to execute the same query multiple times.

We have implemented a query caching mechanism, which is structured into three different components: 1) a mechanism to form a graph structure for a given query which can be programmatically accessed, 2) a mechanism to compare two query graphs, 3) a single eviction strategy.

This query caching mechanism was tested on the two pipeline strategies mentioned in the previous section. When using the cache on regular queries, the performance actually decreases, since the cache has to be updated every time a new source is added, defeating the purpose of this cache.

When performing this caching on the second sub query of each of the query pipeline mechanisms, the results improved. The reason for this is that the chances of a similar sub query being executed again are higher than the chances for a full query, while the time required to update the cache for a given query might be high whenever a new source is added. On the other hand, when it comes to the amount of data that has to be stored, and reusability of each cached result, the first query pipeline mechanism is better, since it provides a more general query to be executed and then stored in the cache. This would not be possible using the second query pipeline mechanism. The use of the first

pipeline mechanism also reduces the overhead that is caused by the second pipeline mechanism, since chances are smaller that duplicate results are stored.

As a result, we can conclude that the query caching strategy is best to be used with one of the query pipeline mechanisms. The recommended mechanism in this case is the first one, for the reasons mentioned before.

8.2 Future work

This section describes future work that can be done in the area of this thesis for both the query pipeline as well as the query caching mechanism.

8.2.1 Query pipeline

We have implemented a query analysis strategy, together with two query pipeline construction strategies and a single strategy to join them. These mechanisms had to be as simple and efficient as possible. Several improvements and extensions can be made to improve these mechanisms or provide new ways for them to be managed.

First, the query analysis strategy could be extended by providing support for more of the types and clauses found in a SPARQL query. At the moment only the basic structure elements, such as triple patterns, union and filter are supported. Next to this, new strategies could be investigated which try to fully automate the analysis, compared to the current system using named graphs.

Secondly, since the query analysis only has basic support for SPARQL queries, the two construction strategies, also only provide basic support. Both of the query construction strategies can be extended to provide support for the rest of the elements found in SPARQL queries.

Thirdly, there is only one strategy currently provided for joining results. Extra strategies could be constructed which provide more efficient ways to join two results together, more specifically based on the construction strategy developed for query construction. The current solution is more used as a solution that works for all, instead of being optimized for various query construction strategies.

8.2.2 Query caching

As was the case before, the query graph building mechanism is only able to take into account the SPARQL basics, such as triple patterns, unions and filters. Currently when a filter is present, the information provided by this filter is not used. This prevents more complex queries from being compared with each other. The current mechanism could be extended to provide support for all the other structures, including better support for filters, available in a SPARQL query, allowing support for more advanced queries. Next to this, the RelationHelper mentioned in section 6.3 can be extended to allow automatic retrieval of (existing) ontologies (i.e. to retrieve information about inverse relationships, subclasses and subproperties).

The query comparison mechanism can be said to have the same limitations. At the moment, it is able to compare any query based on the rules mentioned in section 6.1. This comparison strategy can be extended to add support for structures such as the filter to allow more accurate comparisons to be made. Next to just being able to compare two graphs, this strategy could also be extended to check whether a certain query contains part of the results for the new query. This would allow remainder queries to be formed, as mentioned in section 3.3.2; these allow one part of the query result to be retrieved from the cache, and the other part by executing the portion of the query referencing the

missing cache data. This would result in better cache utilization and performance, as only a small part of the query needs to be re-executed.

Replacement eviction strategies could be created better suited to the person's situation. The currently implemented LRU eviction strategy is only useful in some situations (e.g. only the most recently used information is used often), whereas other eviction strategies can be useful for other situations. For example the second eviction strategy mentioned in section 3.3.4, which is based on the Manhattan distance between two semantic regions. Based on this distance the relevance of the query results available in the cache is determined based on the last executed query. Since our expectations are that a user always wants to know more about the result he is currently querying, this mechanism would remove irrelevant results from the cache (when needed), even when they are more recent than the relevant information stored in the cache. Adding new eviction strategies gives developers the choice to choose which they find the most appropriate, or allow an eviction strategy to be chosen automatically based on the situation a client is in. Next to the eviction strategy, the storage of the cache can also be improved. At the moment, the cache is only stored in volatile memory, making the amount of data that can be cached limited. Adding support for a caching strategy which (at least partially) stores this data in permanent memory (e.g. an SD card) would significantly increase the number of elements that can be stored.

Chapter 9. Bibliography

1. William Van Woensel , Sven Casteleyn, Olga De Troyer, A Framework for Decentralized, Context-Aware Mobile Applications Using Semantic Web technology, 2009.
2. (2011, April) <http://source.android.com>. [Online] <http://source.android.com/about/index.html>
3. (2011, April) <http://en.wikipedia.org/>. [Online] http://en.wikipedia.org/wiki/Google_Android
4. (2009, March) <http://en.wikipedia.org/>. [Online] http://en.wikipedia.org/wiki/File:Diagram_android.png
5. Beat Signer. (2010, Nov.) Web Information Systems: The Semantic Web.
6. (February, 2004) <http://www.w3.org/>. [Online] <http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>
7. (February, 2004) <http://www.w3.org/>. [Online] <http://www.w3.org/TR/rdf-schema/>
8. (February, 2004) <http://www.w3.org/>. [Online] <http://www.w3.org/TR/owl-features/>
9. (2011) <http://en.wikipedia.org>. [Online] <http://en.wikipedia.org/wiki/SPARQL>
10. (January, 2008) <http://www.w3.org/>. [Online] <http://www.w3.org/TR/rdf-sparql-protocol/>
11. (January, 2008) <http://www.w3.org/>. [Online] <http://www.w3.org/TR/rdf-sparql-query/>
12. William Van Woensel, Sven Casteleyn, Elien Paret, Olga De Troyer, Exploiting Existing Online Semantic Web Data and Technologies to Develop Context-Aware, Mobile Applications, 2010 of 2011.
13. Espinoza F., Persson P., Sandin A., Nyström H, Cacciatore E., Bylund M., GeoNotes: Social and Navigational Aspects of Location-Based Information Systems, 2001.
14. López-de-Ipiña, D., Vazquez, J.I., Abaitua, J.: A Context-aware Mobile Mash-up Platform For Ubiquitous Web. In: 3rd IET International Conference on Intelligent Environments, pp. 116--123, IEEE, Ulm, Germany (2007).
15. Challiol, C., Rossi, G., Gordillo, S., De Cristófolo, V.: Designing and Implementing Physical Hypermedia applications. In: ICCSA 2006, UWSI 2006, pp. 148--157, Springer Berlin / Heidelberg (2006).
16. Roduner, C., Langheinrich, M.: Publishing and Discovering Information and Services for Tagged Products. In: 19th International Conference on Advanced Information Systems Engineering, pp.501--515, Springer Berlin / Heidelberg, Trondheim, Norway (2007).
17. Judd, G., and Steenkiste, P. Providing contextual information to pervasive computing applications. In Proceedings of the First IEEE International Conference on Pervasive Computing and Communication. IEEE, 2003, 133-142.
18. Gu, T., Pung, H.K., and Zhang, D.Q. A middleware for building context-aware mobile services. In Proceedings of IEEE Vehicular Technology Conference. IEEE, 2004, 2656-2660.
19. Euzenat, J., Pierson, J., and Ramparany, F. Dynamic context management for pervasive applications. Knowledge Engineering 23, 1, 2008, 21-49.
20. Dennis Quan, David R. Karger, How To Make a Semantic Web Browser, 2004.
21. Baihua Zheng, Wang-Chien Lee, Dik Lun Lee, On Semantic Caching and Query Scheduling for Mobile Nearest-Neighbor Search, 2004.
22. Yigal Arens, Graig A. Knoblock, Intelligent Caching: Selecting, Representing, and Reusing Data in an Information Server, 1994.
23. Stuckenschmidt, H., Vdovjak, R., Houben, G.J., Broekstra, J. Towards Distributed Processing of RDF Path Queries. International Journal of Web Engineering and Technology, 2, 2/3, 2005, 207-230.

24. Quilitz, B., and Leser, U. Querying Distributed RDF Datasources with SPARQL. In Proceedings of the 5th European Semantic Web Conference. Springer, 2008, 524-538.
25. Kaoudi, Z., Kyzirakos, K., and Koubarakis, M. SPARQL Query Optimization on Top of DHTs. In Proceedings of 9th International Semantic Web Conference. Springer, 2010, 418-435.
26. Michael Martin, Jörg Unbehauen, Sören Auer, Improving the Performance of Semantic Web Applications with SPARQL Query Caching.
27. Mengdoing Yang, Gang Wu, Caching Intermediate Results of SPARQL Queries, 2011.
28. Dominic Battré, Caching of intermediate results in DHT-based RDF stores, 2008.
29. Michael Wessel, Ralf Möller, A High Performance Semantic Web Query Answering Engine.
30. S. Kami Makki, Xunhang Zhou, Novel Cache Management Strategy for Semantic Caching in Mobile Environment, 2008.
31. Boris Chidlovskii, Uwe M. Borghoff, Semantic caching of Web queries, 2000.
32. Dongwon Lee, Wesly W. Chu, Semantic Caching via Query Matching for Web Sources, 1999.
33. Renu Tewari, Harrick M. Vis, Asit Dan, Dinkar Sitaram, Resource-based Caching for Web Servers.
34. Jianlian Xu, Xuejan Tang, Dik Lun Lee, Performance Analysis of Location-Dependent Cache Invalidation Schemes for Mobile Environments, 2003.
35. Björn Dór Jónsson, María Arinbjarnar, Bjarnsteinn Dórsson, Performance and Overhead of Semantic Cache management, 2006.
36. Shaul Dar, Michael J. Franklin, Björn T. Jónsson, Divesh Srivastava, Michael Tan, Semantic Data Caching and Replacement, 1996.
37. Parke Godfrey, Jarek Gryz, Semantic Query Caching for Heterogeneous Databases, 1997.
38. Heiner Stuckenschmidt, Similarity-Based Query Caching, 2004.
39. (June, 2011) <http://champignon.net/>. [Online] <http://champignon.net/cooltown.php>
40. Wolfgang De Meuter. (2010, Mar.) Algoritmen en Datastructuren 2.