Vrije Universiteit Brussel

Faculty of Science,
Department of Computer science

Efficient querying of distributed sources in a mobile environment through source indexing and caching.

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

Elien Paret

Promoter: Prof. Dr. Olga De Troyer

Advisors: Dr. Sven Casteleyn & William Van Woensel

Academic year 2009-2010

Acknowledgements

In this small chapter I would like to thank everyone that helped me achieve this thesis. First of all I would like to thank my promoter, Prof. Olga De Troyer, for providing me the opportunity to realize this thesis and for her personal assistance during this entire academic year.

Secondly, I would like to express my deepest gratitude to my two excellent advisors, Dr. Sven Casteleyn and PhD William Van Woensel. I am very thankful for their personal guidance, moral support and all the knowledge they shared with me.

I also want to thank my dear boyfriend Pieter Callewaert for his moral support, kindness, patience, encouragement and for being there for me at all circumstances and times.

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

Thank you all.

Abstract

Mobile devices have become a part of the everyday life; they are used anywhere and at anytime, for communication, looking up information, consulting an agenda, making notes, playing games etc. At the same time, the hardware of these devices has evolved significantly: e.g., faster processors, larger memory and improved connectivity … The hardware evolution along with the recent advancements of identification techniques, has lead to new opportunities for developers of mobile applications: mobile applications can be aware of their environment and the objects in it. Combining these new opportunities with the Web, allows mobile applications to use services and information of nearby objects (e.g., a mobile application that informs you of the restaurants that are nearby your current position without the need to enter your current position).

The SCOUT framework, currently being developed at the WISE lab, supports the development of context- and environment-aware mobile applications, by providing a conceptual and integrated view on the environment called the Environment Model. This Model comprises metadata on physical entities found nearby the user, and the user's own profile information, and thus allows applications to become aware of (and responsive to) the user's physical environment and context. SCOUT is a decentralized and distributed solution, where no single centralized server is required for storing context-sensitive data or integrating data from various sources. Instead, this integration is achieved via the locally maintained Environment Model, while each content provider is responsible for making available and managing their own data. In order to facilitate information integration across different heterogeneous sources, Semantic Web technology is employed.

Until now, the Environment Model has been stored as a fully materialized view; in other words, all of the data (i.e., encountered data sources) is kept locally. This thesis investigates how the Environment Model can be constructed and managed more efficiently. Each of the strategies mentioned below have been tested extensively in different scenarios, in order to determine the most suitable ones and to indicate where there is room for improvement.

Firstly, we have investigated several ways of storing summary information on encountered data sources to determine which sources contain relevant information for a given query. The goal of these strategies is to avoid having to include all encountered sources when answering a query.

Secondly, we have developed several caching strategies where some data from encountered sources is kept locally, to avoid having to download a relevant sources (as identified by one of the strategies mentioned above) every time it is needed to solve a query issued to the Environment Model.

Samenvatting (in Dutch)

Mobiele toestellen zijn de afgelopen jaren een belangrijk onderdeel geworden van het dagelijkse leven; ze worden overal en te allen tijde gebruikt om te communiceren, te chatten, informatie op te zoeken, een agenda te raadplegen, notities te maken, spelletjes te spelen, enz. Terzelfder tijd is de hardware van deze mobiele toestellen sterk geëvolueerd: snellere processors, groter geheugen en verbeterde verbindingsmogelijkheden … De combinatie van deze hardware evolutie, samen met de recente vooruitgangen in identificatie technologieën (bijv., RFID, NFC) hebben het mogelijk gemaakt mobiele applicaties te ontwikkelen die bewust zijn van hun omgeving en de zaken die zich in hun omgeving bevinden. Door deze nieuwe vooruitgangen te combineren met het Web, kunnen mobiele applicaties online services en informatie uitbuiten van zaken, objecten, enz in hun omgeving (bijv., mobile applicaties die winkels weergeven die producten verkopen gerelateerd aan de gebruiker’s profiel en/of bezienswaardigheden dat de gebruiker die dag bezocht heeft).

Momenteel wordt op het WISE lab het SCOUT framework ontwikkeld. Dit framework ondersteunt de ontwikkeling van mobiele applicaties die zich bewust zijn van de context en de omgeving van de gebruiker. Dit gebeurt door middel van een conceptueel data model, genaamd het Environment Model, dat de gebruiker’s eigen profiel informatie omvat samen met meta-data van fysische objecten uit de gebruiker‘s omgeving. Dit model laat toe dat applicaties zich bewust worden van en kunnen reageren op de fysieke omgeving en context van de gebruiker. SCOUT is een gedecentraliseerde en gedistribueerde oplossing, waarin geen enkele centrale server vereist is om de contextgevoelig data op te slaan of om de data van verschillende bronnen samen te voegen. In plaats daarvan zorgt het lokaal bijgehouden Environment Model voor de integratie van de verschillende bronnen, terwijl elke content provider verantwoordelijk is om zijn eigen informatie ter beschikking te stellen en te onderhouden. Om de integratie van deze verschillende heterogene bronnen mogelijk te maken wordt gebruik gemaakt van Semantische Web technologie.

Tot nu toe werd alle gevonden informatie (d.i., het Environment Model) lokaal bewaard. Deze thesis onderzoekt hoe dit Environment Model efficiënter kan worden aangemaakt en bijgehouden. Elk van de onderstaande strategieën zijn uitgebreid getest in verschillende scenario’s, om te bepalen welke de meest geschikte is en alsook te bepalen waar er ruimte is voor verbetering.

Eerst hebben we onderzocht welke samenvattende informatie moet bijgehouden worden voor elke ontdekte bron om te kunnen bepalen welke bron relevante informatie bevat om een gegeven query op te lossen. Door middel van deze samenvattende informatie (d.i., het Source Index

Model) wordt vermeden dat we steeds alle ontdekte bronnen moeten includeren om een query op te lossen.

Ten tweede hebben we verschillende caching strategieën ontworpen die de data uit ontdekte bronnen lokaal bewaren. Dit zorgt ervoor dat relevant bronnen (geïdentificeerd door één van bovenvermelde source index strategieën) niet steeds opnieuw moeten gedownload worden wanneer een query gesteld wordt aan het Environment Model.

Table of content

List of charts

List of figures

List of RDF Data

List of SPARQL Queries

List of tables

Glossary of terms

The **actual domain** of a predicate specifies the type of resource that actually occurs as subject of that predicate

Cache Strategy is the part of the SCOUT environment layer which is responsible for determining what information of the encountered sources is kept locally.

The **domain** of a predicate specifies the type of resource that can occur as subject of that predicate.

The **Environment Layer** of the SCOUT framework is responsible for storing and integration information or data about the entity and its current environment (e.g., data of encountered entities).

The **Environment Manager** is a component of the Environment Layer which is responsible for maintaining and providing access to the Environment Model.

The **Environment Model** is the part of the SCOUT framework that comprises the mobile user's own profile information, the past and current positional relationships with other entities and the data that the other entities offer.

Eviction strategy, also called a replacement strategy, is a strategy which is used by a cache to determine what cache elements are evicted from the cache when it is full.

A **given query** is a query posed to the Environment Model.

The **Source Index Model** is a part of the SCOUT environment layer. The Source Index Model is a model which contains information, called summary information, about what information (e.g., predicates and domains …) occurs in the encountered sources.

Source Index Strategy is the strategy responsible for maintaining the Source Index Model.

A **Source Index Query** is a query posed to the Source Index Model with the goal of finding sources that are relevant, in solving another (original) query.

The **object** is the part of an RDF triple that represents the value of the relationship with the subject.

The **predicate**, often called a property, is the part of an RDF triple which indicates the relationship between the subject and object of the triple in which it is used.

RDF (Resource Description Framework) is used to describe a data model in such a way that it facilitates sharing and interchange of data. Different RDF representations are available: RDF/XML, N-Triple, Turtle …

RDF Schema is a technology used to define taxonomies of classes and properties and simple domain and range specifications for properties.

An **RDF triple**, often called a statement, consists of a subject, predicate and object. It describes the relationship, indicated by the predicate, between the subject and object of the triple.

RDF/XML is an XML serialization of a set of RDF triples.

Reification makes it possible to uniquely identify a statement to be able to refer to it and its parts (e.g., the predicate). This is useful when extra information about the statement itself must be stored (e.g., when the statement was created).

The **SCOUT framework** is a framework that supports the development of context-aware mobile applications. Depending on the mobile user's environment and needs at a given time and place, these mobile applications offer relevant information and services.

SPARQL is the W3C recommendation query language and protocol used to access RDF information.

The **subject** is the part of an RDF triple which indicates the resource for on which a relationship, property, is expressed.

This page is intentional left blank.

Chapter 1 Introduction

This first chapter describes the context of this thesis, a description of the problem it solves, the approach that is followed to solve the problem and the structure of this thesis.

1.1 Context

Mobile devices have become a part of the everyday life; they are used anywhere and at anytime, for communication, looking up information, consulting an agenda, making notes, playing games, etc. At the same time, the hardware of these devices has evolved significantly, including faster processors, larger memory and improved connectivity. Although this evolution has allowed mobile devices to lessen the gap with desktop computers, both in functionality and user-friendliness, mobile devices still have some limitations. For one, the input and display capabilities of these devices are still lacking, making it cumbersome to enter information and get a comprehensive information overview. Secondly, because the device is deployed in a mobile environment, the user cannot spend the same amount of time looking for information as when sitting behind a desk: he is walking around, minding traffic, sitting at a bar, etc.

However, it can be observed that in a mobile environment, the user is often interested in information related to his current situation: for instance, reviews of nearby restaurants, products (souvenirs) from nearby shops related to previously visited monuments, etc. Consequently, by automatically taking into account the user's current and past environment and context when looking up this information, the user can be relieved of much of the work involved. Identification technology (e.g., RFID, NFC) can be employed to make mobile applications aware of the user's physical environment and the entities in it; combined with the improved connectivity of mobile devices, this allows mobile applications to access services and information from the Web associated with nearby objects.

The SCOUT framework, currently being developed at the WISE lab, supports the development of context- and environment-aware mobile applications, by providing a conceptual and integrated view on the environment called the Environment Model. This model includes the data associated with currently (and previously) nearby physical entities, together with the user's own profile information, and thus allows applications to become aware of (and responsive to) the user's physical environment and context.

SCOUT is a decentralized and distributed solution, where no single centralized server is required for storing data on the user's environment or integrating data from the various data sources, (associated with the environment entities). Instead, this integration is achieved via the Environment Model, which is locally maintained, while each content provider is responsible for providing and managing their own data. In order to facilitate information integration of the different heterogeneous sources, Semantic Web technology is used.

1.2 Problem description

The Environment Model provides the data associated with currently (and previously) nearby physical entities, together with the user's own profile information. By providing such a model, mobile applications can become aware of (and responsive to) the user's physical environment and context. Until now, this model has been stored as a fully materialized view; in other words, all of the data (i.e., encountered data sources) is stored locally.

A first drawback of this approach is that it is unrealistic to store all the data (related to encountered entities) locally, since mobile devices have only a limited amount of space available. Secondly, when accessing this Environment Model (i.e., via a query), all of the discovered data is used to solve a given query, despite the fact that a large part of the data will be irrelevant for that query. In order to determine what sources are relevant for a certain query, we must firstly know what information is contained in our heterogeneous sources; secondly, we must know what information the query needs.

This thesis investigates how the Environment Model can be constructed and managed more efficiently in order to avoid these two drawbacks.

1.3 Approach

Until now the Environment Model has been stored as a fully materialized view, meaning all of the data (belonging to encountered entities) is stored locally. In our approach, we construct and manage the Environment Model more efficiently. For this purpose, we have developed two mechanisms: a mechanism which constructs and maintains a Source Index Model, which summarizes the information found in the encountered data sources, and a caching strategy, which decides which data is stored locally. Both mechanisms are described in the following paragraphs.

Firstly, we construct and maintain a Source Index Model, which summarizes the information found in the encountered data sources. The goal of this model is to identify which sources are relevant for a given query, and thus allows us to avoid having to include all encountered data sources when a query needs to be answered. This model is created and extended at runtime: each time a new source is encountered, summary information on that source is extracted and added to the index model. Therefore, the model must be flexible enough to summarize a variety of heterogeneous data sources.

Secondly, we employ a caching strategy, in order to avoid downloading the relevant sources every time a query needs to be answered. Again, this cache strategy should be flexible enough to deal with data sources that contain a wide variety of information and vary in size. While developing this caching strategy, the small memory size and limited performance of mobile devices should be taken into consideration.

We have developed several alternative strategies for each of these optimizations, varying both in amount (Source Index Model) and granularity (cache) of the data kept. These strategies have been validated and compared through extensive experiments.

1.4 Thesis structure

This **first chapter** explained the context, the problem and followed approach of this thesis.

The **second chapter**, which provides background information related to this thesis. It explains in detail topics like the SCOUT framework, Semantic Web, caching …

The **third chapter** describes the related work in scope of this thesis (i.e., the SCOUT framework, Source Indexes and caching strategies), what work has been done previously and how does it differ from the followed approach.

The **fourth chapter** provides an architectural overview.

The **fifth chapter** explains the concept of the Source Index Model, a model providing what kind of information is used in the encountered data sources.

The **sixth chapter** explains the Cache that is used to store some of the encountered data source locally.

The **seventh chapter** provides the implementation details of all the developed components.

Chapter eight describes the experiments and their outcomes that were conducted regarding the Source Index Model and the cache strategy.

Chapter nine elaborates on future work.

C**hapter ten** describes the conclusions of this thesis.

Chapter 2 Background

The sections of this background chapter discuss all the technologies necessary to understand the following chapters. Technologies like Semantic Web, RDF, and RDF Schema … are discussed. The first topic is the SCOUT framework, the framework for which I made this thesis.

2.1 SCOUT framework

This section describes the SCOUT framework for which this thesis is made. Firstly a small description of the framework is given, what the characteristics are. Secondly, an overview of the layered structure of the framework is given; each layer and its most important components are described in detail.

2.1.1 Introduction

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

The SCOUT framework is scalable, decentralized and distributed in which each identifiable entity is responsible for providing and managing its own data and services in the form of a Web presence. This can be a simple website, Web services or an online sources 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. By using Semantic Web standards and vocabularies to describe Web presences in a uniform and expressive way, the SCOUT framework allows seamless integration and querying of data from (several) different entities, thereby providing mobile applications with a richer and more complete view on the global environment.[1]

2.1.2 An Overview

From the introduction we know that the SCOUT framework is framework which supports the development of context-aware mobile applications. The SCOUT framework is a four layered model, which separates the different design concerns and offers independence between layers and the underlying technology. Figure 1 shows the conceptual model of the SCOUT framework and its four layers: the detection layer, the location management layer, the environment layer, the application layer.[1]

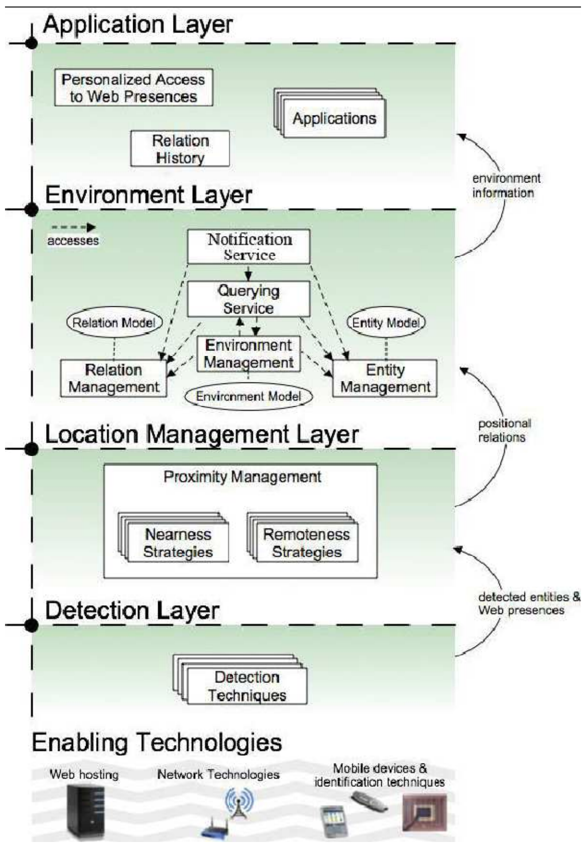


Figure 1: SCOUT, a layered framework [1]

2.1.2.1 Detection layer

The bottom layer, the detection layer is responsible for detecting identifiable physical entities (e.g., a Smartphone) and obtaining the reference to the corresponding Web presence. Each physical entity has a Web presence which contains information about the entity (e.g., a FOAF file). It encapsulates the different detection techniques (e.g., RFID, NFC, Bluetooth, etc) that can be used and extracts references to other Web presences, so that they can be used by other layers.

2.1.2.2 Location management layer

The location management layer is on top of the detection layer and is responsible for creating and maintaining positional relations between different entities. It uses the information from the detection layer to determine which entities are nearby the user and which are no longer nearby the user. To determine whether or not an entity is nearby or no longer nearby so-called "*nearness*" and "*remoteness*" strategies are used, which are collectively called proximity strategies. A "*positional*" relation is used to express the "*nearbyness*" of an entity.

2.1.2.3 Environment layer

The Environment layer stores and integrates information or data about the entity and its current environment. It also provides services to obtain information from nearby Web presences in both a push- and pull-based manner. Semantic Web tools like Jena[1] are used to store the data, integration of the data happens via RDF[2], a Semantic Web technology, and the reasoning capabilities of the Semantic Web are used to infer new information. [1]

A Relation model is used to express the "*positional*" relation between the entity and the entities it has met. An Entity model is used for representing the metadata or information of an entity. Both models (Relation model and Entity model) have a management component which maintains the model, allows querying, programmatic access and provides views of the model.

The Query Service is the core component of the Environment layer; it allows client applications to query the Entity model, the Relation model and the models of Web presences of (nearby) entities. These queries can vary from simple queries retrieving metadata to queries containing complex semantic conditions (e.g. references to the user's Entity Model).[1]

The Notification Service allows applications to obtain references of nearby entities in a push-based manner, thus allowing them to become responsive to changes in the mobile user's environment. An application has the possibility to register itself with this service in order to automatically receive a notification (in the form of an event) when nearby entities are encountered or are no longer nearby. Per default, the application receives a notification of all nearby entities. Additionally, the framework allows filtering by enabling the application to specify a condition which must be satisfied by the nearby entity before the application is notified.[1]

The Environment Management component is responsible for combining metadata from multiple Web presences, obtained from the past and current positional relations (Relation Model), and integrating it with metadata of the mobile user (Entity Model). The combination of all this information is called the Environment Model. Notice that posing a query to the Environment Model, actually causes different sources to be queried. This is a major benefit for the developer, since he does not need to take into consideration that he is actually querying several different Web presences, the Entity Model and Relation Model.

The top layer is the application layer, which contains applications that have access to the services and models provided by the framework and thus can take the user's environment (current and previous) into consideration.[1]

[1] http://jena.sourceforge.net/

[2] http://www.w3.org/RDF/

2.2 Semantic Web

As explained in the previous section, the SCOUT framework is a framework that supports the development of context-aware mobile applications. Depending on the mobile user’s environment and needs at a given time and place, these mobile applications offer relevant information and services. [1] In order to accomplish his goals, SCOUT uses Semantic Web technologies like RDF and Semantic Web tools like Jena. It is thus necessary to explains what the Semantic Web is, the idea behind it, the purpose of it, as well as a layered conceptual model of the Semantic Web to show the composition of it.

2.2.1 The idea and purpose of the Semantic Web

The Semantic Web is the further development of the World Wide Web; it can be seen as an extension of the WWW. The purpose is to add semantics or meaning to documents, services, data … on the Web so that all the information becomes machine readable and understandable. This makes it possible for the machine to derive new facts, detect contradictions, and infer new relationships … The search engines benefit from the Semantic Web, since they will be able to return more relevant results. Since the results depend on the actual web content and / or semantics instead of depending on for example keywords that the developers set or on the internal link structure.

The figure below shows the concept of the World Wide Web and how the Semantic Web extends it. Machines in the WWW were simply designed to relay, to display information, not to be aware of the meaning, concepts and relationships contained in it. This makes it very difficult for applications to utilize the WWW as an information source in an automated way. While with the Semantic Web, the machines are aware of the concepts and relationships that are present. [2]

The figure below shows that the Semantic Web does not change anything to the distributed manner of the World Wide Web today. But by using ontologies[3], adding concepts and adding relationships to the information on the Web, the expressiveness of the information is increased.

[3] An ontology is a formal representation of a set of concepts within a domain and the relationships between those concepts. [27]

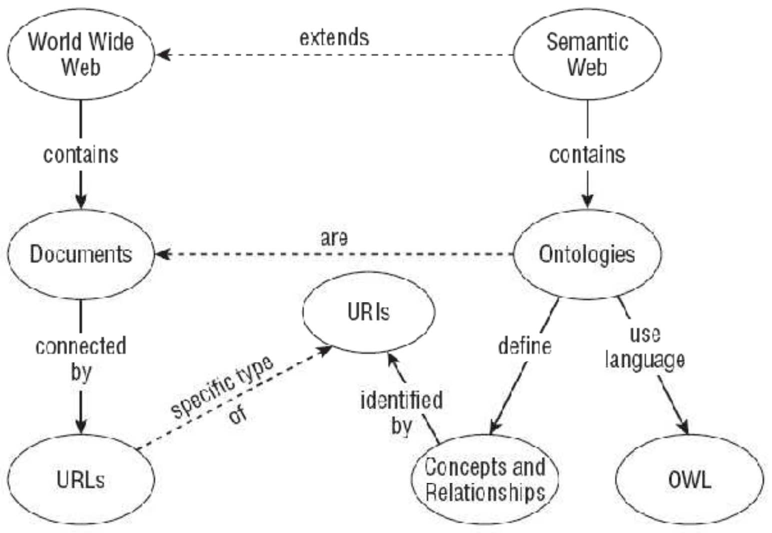


Figure 2: The concept of the WWW and the Semantic Web [2]

As mentioned previously, the Semantic Web adds concepts and relations to the information on the Web. Relationships, in fact, take on a primary role in the Semantic Web. Object-oriented solutions make relationships secondary to the objects themselves. Relationships do not exist outside of an object. Relationships are dependent on their associated object class and cannot be repurposed for other classes. Relationships in the Semantic Web exist distinct from the concepts that they join, they are first-class. Relationships are free to join any collection of statements. This allows relationships to have inheritance and restriction rules of their own. For example, a social network relationship within the Semantic Web could offer an "*associatedWith*" relationship that contains a sub relationship "*ownsBy*" and another sub relationship "*friendOf*".[2]

2.2.2 The Semantic Web Stack

The Semantic Web is represented by a conceptual layered model, called the Semantic Web Stack. Figure 3 is a common figure to represent the Semantic Web stack, it shows the fourteen layers:

- The URI / IRI layer is used to uniquely identify Semantic Web Sources.
- The Unicode layer can represent and manipulate information in different languages.
- The XML layer is used to interchange structured data over the web.
- Namespaces integrate markup from multiple sources.
- XML Query queries collections of XML data.
- XML Scheme defines the structure of specific XML languages.
- The RDF Model & Syntax layer represents resource information as a graph and describes taxonomies based on RDFS.
- The Ontology layer is for defining vocabularies and enabling reasoning based on description logic.
- Rules / Query describe addition rules via RIF and queries RDF data based on the SPARQL query language.

- The logic layer is for logical reasoning, to infer new facts and check consistency.
- Proof explains the logical reasoning steps taken.
- Trusts is authentication and trustworthiness of derived facts based on Proof, Signature and Encryption.
- Signature validates the source of facts by digitally signing RDF data.
- Encryption protects RDF Data via encryption.

[3]

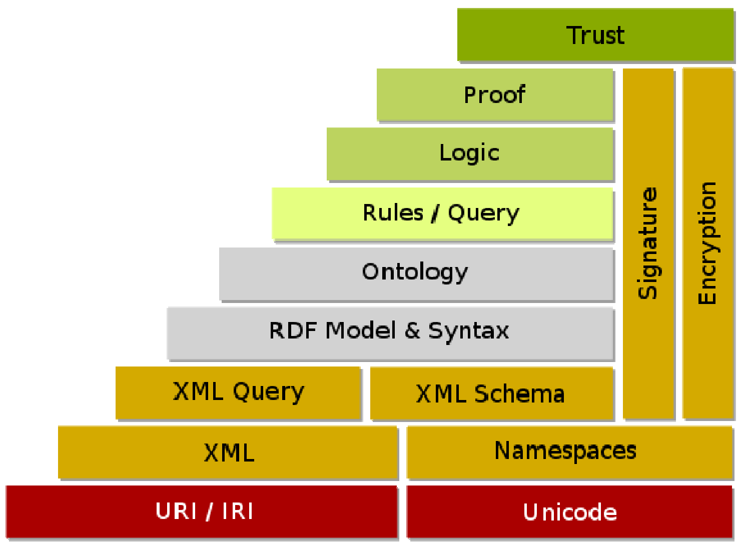


Figure 3: The Semantic Web Stack [4]

2.3 Resource Description Framework

The detection layer of the SCOUT framework can detect various entities (e.g., a mobile phone, a building …); these various entities share information with the main entity which is in the RDF format. This section explains what RDF (Resource Description Framework) is and how it can be represented.

2.3.1 What

The Semantic Web Stack has an RDF Model & Syntax layer. This layer is responsible for representing resource information as a graph and to describe taxonomies and vocabularies based on RDFS. RDF is responsible for doing the former namely describing the data model in such a way that it facilitates sharing and interchange of data.

2.3.2 Representation

RDF information is represented by so-called triples. A triple is of the form “*<subject> <property> <object>*”. A subject and property are a resource, while an object is either a

Resource or a Literal. A Resource is a concept or object which can be uniquely identified by an IRI[4] (e.g. an URL). A Literal represents concrete data values like numbers, strings …

The subject of a triple is the Resource of which we want to say something; the property is the relation between the subject and the object and the object is the value of the property for the subject. In the example below "`http://wise.vub.ac.be/Elien#me`" is the subject, "`http://xmlns.com/foaf/0.1/nick`" and "`http://xmlns.com/foaf/0.1/age`" are the properties and "`Elien`" and "`23`" are the objects, both are Literals.

```
<http://wise.vub.ac.be/Elien#me> <http://xmlns.com/foaf/0.1/nick> "Elien" .
<http://wise.vub.ac.be/Elien#me> <http://xmlns.com/foaf/0.1/age> 23 .
```

RDF data 1: Example of a RDF triple

RDF information is commonly presented, visualized as a labeled, directed graph. Figure 4 is an example of RDF information visualized as a labeled directed graph. This graph is thus a collection of triples, a collection of IRIs and Literals that all are unique. That uniqueness combined with the fact that a RDF graph has no actual root facilitates the merging of two or more RDF graphs. Merging a RDF graph means just putting them next to each other, there is no need for name translation.

Although graphs are a very powerful tool for representing information, they are unsuitable for exchanging them between several applications. To easily exchange RDF information between applications, there exist several RDF serializations (e.g., RDF/XML, Turtle and N-Triple). [2]

RDF/XML is an XML application for representing RDF triples in a serialized way, it is the only normative (standard) exchange syntax for RDF serialization. RDF data 2 shows the RDF/XML representation of the same RDF information as in Figure 4.

The Terse RDF Triple language, or Turtle, is not an XML language but a more human-friendly and a more readable syntax since it does not use a tree to represent the RDF information. RDF data 3 shows the same RDF information as in RDF data 2 and Figure 4 but in the TURTLE format.

N-Triples is a simplified version of Turtle. N-Triples uses the same syntax for comments, URIs and Literal values but has some simplifying restrictions. For example in Turtle the following statements are valid: "`subject predicate1 value1; predicate2 value2 .`", one can use the ";" shorthand to use the subject of the previous statement. In N-Triple this is not allowed, a statement in N-Triple is represented by a single line containing the subject, predicate and object. This

[4] Internationalized Resource Identifier

simplified form makes it an attractive choice for serializing RDF, particularly in applications with streaming data. [2]

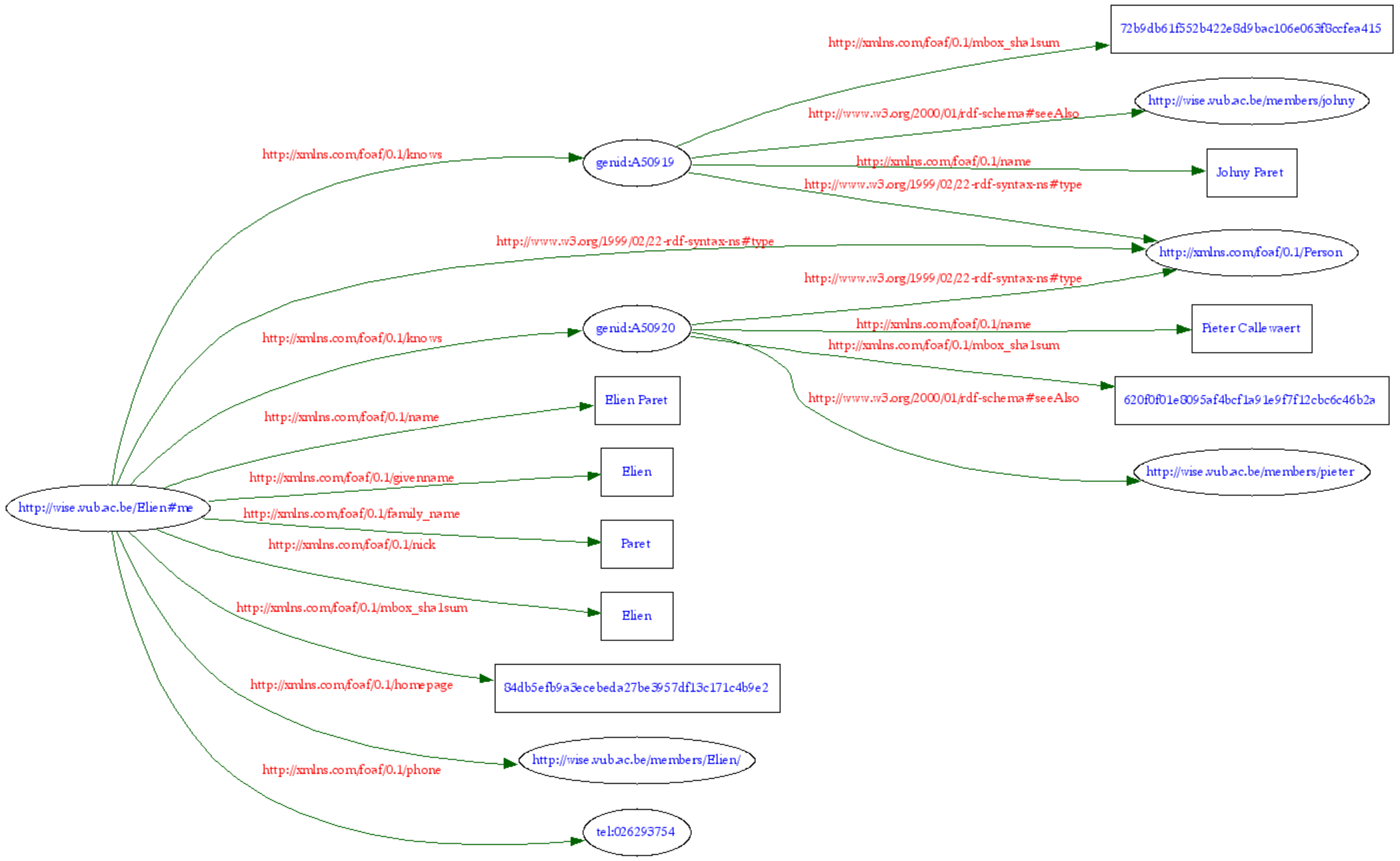

http://wise.vub.ac.be/Elien#me
http://xmlns.com/foaf/0.1/knows
genid:A50919
http://xmlns.com/foaf/0.1/mbox_sha1sum
72b9db61f552b422e8d9bac106e063f8ccfea415
http://www.w3.org/2000/01/rdf-schema#seeAlso
http://wise.vub.ac.be/members/johny
http://xmlns.com/foaf/0.1/name
Johny Paret
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://xmlns.com/foaf/0.1/Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://xmlns.com/foaf/0.1/knows
genid:A50920
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://xmlns.com/foaf/0.1/name
Pieter Callewaert
http://xmlns.com/foaf/0.1/mbox_sha1sum
620f0f01e8095af4bcf1a91e9f7f12cbc6c46b2a
http://www.w3.org/2000/01/rdf-schema#seeAlso
http://wise.vub.ac.be/members/pieter
http://xmlns.com/foaf/0.1/name
Elien Paret
http://xmlns.com/foaf/0.1/givenname
Elien
http://xmlns.com/foaf/0.1/family_name
Paret
http://xmlns.com/foaf/0.1/nick
Elien
http://xmlns.com/foaf/0.1/mbox_sha1sum
84db5efb9a3ecebeda27be3957df13c171c4b9e2
http://xmlns.com/foaf/0.1/homepage
http://wise.vub.ac.be/members/Elien/
http://xmlns.com/foaf/0.1/phone
tel:026293754

Figure 4: Example RDF Graph

```
<?xml version='1.0'?>
<rdf:RDF
      xml:base="http://wise.vub.ac.be/Elien"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:foaf="http://xmlns.com/foaf/0.1/">
<foaf:Person rdf:ID="me">
   <foaf:name>Elien Paret</foaf:name>
   <foaf:givenname>Elien</foaf:givenname>
   <foaf:family_name>Paret</foaf:family_name>
   <foaf:nick>Elien</foaf:nick>
   <foaf:mbox_sha1sum>84db5efb9a3ecebeda27be3957df13c171c4b9e2</foaf:mbox_sha1sum>
   <foaf:homepage rdf:resource="http://wise.vub.ac.be/members/Elien/"/>
   <foaf:phone rdf:resource="tel:026293754"/>
   <foaf:knows>
      <foaf:Person>
         <foaf:name>Johny Paret</foaf:name>
         <foaf:mbox_sha1sum>72b9db61f552b422e8d9bac106e063f8ccfea415>/foaf:mbox_sha1sum>
         <rdfs:seeAlso rdf:resource="http://wise.vub.ac.be/members/johny"/>
      </foaf:Person>
   </foaf:knows>
   <foaf:knows>
      <foaf:Person>
         <foaf:name>Pieter Callewaert</foaf:name>
         <foaf:mbox_sha1sum>620f0f01e8095af4bcf1a91e9f7f12cbc6c46b2a</foaf:mbox_sha1sum>
         <rdfs:seeAlso rdf:resource="http://wise.vub.ac.be/members/pieter" />
      </foaf:Person>
   </foaf:knows>
</foaf:Person>
</rdf:RDF>
```

RDF data 2: RDF information in RDF/XML format

```
_:A3aa10439X3aX1271e1f9360X3aXX2dX7fff <http://www.w3.org/2000/01/rdf-schema#seeAlso> <http://wise.vub.ac.be/members/pieter> .
_:A3aa10439X3aX1271e1f9360X3aXX2dX7fff <http://xmlns.com/foaf/0.1/mbox_sha1sum> "620f0f01e8095af4bcf1a91e9f7f12cbc6c46b2a" .
_:A3aa10439X3aX1271e1f9360X3aXX2dX7fff <http://xmlns.com/foaf/0.1/name> "Pieter Callewaert" .
_:A3aa10439X3aX1271e1f9360X3aXX2dX7fff <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
<http://wise.vub.ac.be/Elien#me> <http://xmlns.com/foaf/0.1/nick> "Elien" .
<http://wise.vub.ac.be/Elien#me> <http://xmlns.com/foaf/0.1/mbox_sha1sum> "84db5efb9a3ecebeda27be3957df13c171c4b9e2" .
<http://wise.vub.ac.be/Elien#me> <http://xmlns.com/foaf/0.1/knows> _:A3aa10439X3aX1271e1f9360X3aXX2dX8000 .
<http://wise.vub.ac.be/Elien#me> <http://xmlns.com/foaf/0.1/knows> _:A3aa10439X3aX1271e1f9360X3aXX2dX7fff .
```

```
<http://wise.vub.ac.be/Elien#me> <http://xmlns.com/foaf/0.1/homepage> <http://wise.vub.ac.be/members/Elien/> .
<http://wise.vub.ac.be/Elien#me> <http://xmlns.com/foaf/0.1/name> "Elien Paret" .
<http://wise.vub.ac.be/Elien#me> <http://xmlns.com/foaf/0.1/phone> <tel:026293754> .
<http://wise.vub.ac.be/Elien#me> <http://xmlns.com/foaf/0.1/givenname> "Elien" .
<http://wise.vub.ac.be/Elien#me> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
<http://wise.vub.ac.be/Elien#me> <http://xmlns.com/foaf/0.1/family_name> "Paret" .
_:A3aa10439X3aX1271e1f9360X3aXX2dX8000 <http://www.w3.org/2000/01/rdf-schema#seeAlso> <http://wise.vub.ac.be/members/johny> .
_:A3aa10439X3aX1271e1f9360X3aXX2dX8000 <http://xmlns.com/foaf/0.1/mbox_sha1sum> "72b9db61f552b422e8d9bac106e063f8ccfea415" .
_:A3aa10439X3aX1271e1f9360X3aXX2dX8000 <http://xmlns.com/foaf/0.1/name> "Johny Paret" .
_:A3aa10439X3aX1271e1f9360X3aXX2dX8000 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
```

RDF data 3: RDF information in TURTLE format

The strength of RDF lies in the fact that it is possible to make a statement about anything, even other statements. When making a statement B about a statement A, a resource must be used to represent statement A, since a RDF statement must always have a resource as a subject. Declaring a statement as a resource and using the resource is called reification, the type of this resource is `rdf:Statement`. This resource has three properties: `rdf:subject`, `rdf:predicate` and `rdf:object` defining the statement. [2]

2.4 RDF Schema

RDF Schema (RDFS) is the technology which is used in this thesis to specify an own vocabulary. RDFS is used to define taxonomies of classes and properties and simple domain and range specifications for properties. When RDF is used alone, it lacks support for expressing meaning, or semantics, behind the descriptions. Using RDFS, classes and properties can be arranged in generalization / specialization hierarchies, define domain and range expectations for properties, assert class membership, define collections, specify and interpret data types...[2]

The example below shows the definition of four classes (“`Vehicle`”, “`Car`”, “`Bicycle`” and “`Color`”) and one property (“`hasColor`”). By using the RDF Schema below, every “`Vehicle`” (whether it is a “`Car`” or “`Bicycle`”) will be able to have a property “`Color`”.

```
<?xml version='1.0'?>
<rdf:RDF
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
<rdfs:Class rdf:ID="Vehicle">
   <rdfs:comment>Vehicle Class</rdfs:comment>
   <rdfs:subClassOf rdf:resource="rdf;Resource" />
</rdfs:Class>

<rdfs:Class rdf:ID="Car">
   <rdfs:comment>Car Class</rdfs:comment>
   <rdfs:subClassOf rdf:resource="#Vehicle" />
</rdfs:Class>

<rdfs:Class rdf:ID="Bicycle">
   <rdfs:comment>Bicycle Class</rdfs:comment>
   <rdfs:subClassOf rdf:resource="#Bicycle" />
</rdfs:Class>

<rdfs:Class rdf:ID="Color">
   <rdfs:comment>Color Class </rdfs:comment>
   <rdfs:subClassOf rdf:resource="rdf;Resource" />
</rdfs:Class>
<rdf:Property rdf:ID="hasColor">
   <rdfs:comment>The Color of a Vehicle</rdfs:comment>
   <rdfs:domain rdf:resource="#Vehicle" />
   <rdfs:range rdf:resource="#Color" />
</rdf:Property>
</rdf:RDF>
```

RDF data 4: RDF Schema Example

2.5 Web Ontology Language

OWL is the abbreviation of Web Ontology Language. OWL is a language which describes ontologies for the Semantic Web. The difference with RDF Schema is that RDF Schema only supports the basic elements for defining an ontology (e.g., domain, range, isDefinedBy …) while OWL can be used to define a full ontology (e.g., cardinality, enumerated classes, disjointness) with maximum expressiveness. There exist three version of the OWL language: OWL Full, OWL DL and OWL Lite. [2]

OWL Lite supports the need for a classification hierarchy and simple constraints. For example, while it supports cardinality constraints, it only permits cardinality values of 0 or 1. It is simpler to provide tool support for OWL Lite than for its more expressive relatives. OWL Lite provides a quick migration path for thesauri and other taxonomies. It also has a lower formal complexity than OWL DL.

OWL DL supports those users who want the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs, but they can be used only under certain restrictions (for example, while a class may be a subclass of many classes, a class cannot be an instance of another class). The name OWL DL comes from its correspondence with description logics, a field of research that has studied the logics that form the formal foundation of OWL. [2]

OWL Full is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. For example, in OWL Full a class can be treated simultaneously as a collection of individuals and as an individual in its own right. OWL Full allows an ontology to augment the meaning of the pre-defined (RDF or OWL) vocabulary. It is unlikely that any reasoning software will be able to support complete reasoning for every feature of OWL Full. [5]

Every OWL document consists of an optional ontology header, annotations, class and property definitions, and facts about individuals and data type definitions.

An example of an OWL document can be found below. This example is again about Vehicles. While with the RDFS example of the previous section the Vehicles could have as many colors as they want, in the OWL example they can have at most four colors.

```
<?xml version='1.0'?>
<!DOCTYPE rdf:RDF [
  <!ENTITY owl "http://www.w3.org/2002/07/owl#">
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
]>

<rdf:RDF
   xmlns:rdf="&rdf;"
   xmlns:rdfs="&rdfs;"
   xmlns:owl="&owl;"
   xmlns:xsd="&xsd;"
>

<owl:Class rdf:ID="Vehicle">
   <rdfs:comment>Vehicle Class</rdfs:comment>
   <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#hasColor"/>
        <owl:minCardinality
         rdf:datatype="&xsd;NonNegativeInteger">4</owl:minCardinality>
      </owl:Restriction>
   </rdfs:subClassOf>

</owl:Class>

<owl:Class rdf:ID="Car">
   <rdfs:comment>Vehicle Class</rdfs:comment>
   <rdfs:subClassOf rdf:resource="rdf;Resource" />
</owl:Class>

<owl:Class rdf:ID="Car">
   <rdfs:comment>Car Class</rdfs:comment>
   <rdfs:subClassOf rdf:resource="#Vehicle" />
</owl:Class>

<owl:Class rdf:ID="Bicycle">
   <rdfs:comment>Bicycle Class</rdfs:comment>
   <rdfs:subClassOf rdf:resource="#Bicycle" />
</owl:Class>

<owl:Class rdf:ID="Color">
   <rdfs:comment>Color Class </rdfs:comment>
   <rdfs:subClassOf rdf:resource="rdf;Resource" />
</owl:Class>

<owl:ObjectProperty rdf:about="#hasColor">
   <rdf:type rdf:resource="&owl;FunctionalProperty" />
   <rdfs:comment>The Color of a Vehicle</rdfs:comment>
   <rdfs:domain rdf:resource="#Vehicle" />
   <rdfs:range rdf:resource="#Color" />
</owl:ObjectProperty>

</rdf:RDF>
```

RDF data 5: OWL Example

There exist ontology registries, repositories (e.g. Tones Ontology Repository[5]) and search engines (e.g. Swoogle engine[6]), which have as purpose to encourage the reuse of the ontologies, this is one of the most important pillars of the Semantic Web. Some commonly used ontologies are Friend-Of-A-Friend[7], Dublin Core Metadata Initiative[8], and GeoRSS[9] … [2]

2.6 SPARQL

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

2.6.1 What and usage

SPARQL[10], a recursive acronym for SPARQL Protocol and RDF Query Language, is a query language for RDF. There exist other query languages like RDQL[11] (RDF Data Query Language) and SeRQL[12] (Sesame RDF Query Language), only the SPARQL query language is described in detail since the SCOUT framework only (currently) uses SPARQL, it is W3C standardization and has a wide community support as well.

Note that SPARQL is both a query language and a protocol. Most people focus on the query language since it defines the syntax of the queries. The protocol is used to describe how a SPARQL client talks to a SPARQL endpoint/processor (e.g., Virtuoso[13]) both in an abstract sense and using a concrete implementation based on WSDL 2.0[14].[2]

2.6.2 Syntax

There exist four different SPARQL Query forms: SELECT, CONSTRUCT, ASK and DESCRIBE, each of them are explained in detail in the following paragraphs. A drawback of SPARQL is that it has no support for query forms that change data (e.g., CREATE, DELETE, DROP and INSERT in SQL), sub queries and aggregate functions (e.g., SUM, COUNT).

The SELECT keyword instructs endpoints to bind RDF terms (blank nodes, IRIs, or Literals) to variables based on the given graph pattern (e.g. to the WHERE clause). Bindings are simply returned

5 http://owl.cs.manchester.ac.uk/repository/
6 http://swoogle.umbc.edu/
7 http://www.foaf-project.org/
8 http://dublincore.org/
9 http://www.georss.org/Main_Page
10 http://www.w3.org/TR/rdf-sparql-query/
11 http://www.w3.org/Submission/RDQL/
12 http://www.openrdf.org/doc/sesame/users/ch06.html
13 http://dbpedia.org/sparql
14 Web Service Description language

and are not part of an RDF graph. These bindings can be easily displayed in tabular form. SPARQL query 1 shows a select query which returns the name of all "`foaf:Person`" objects.

```
SPARQL Query:
-------------
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?name
WHERE
{
   ?person rdf:type foaf:Person .
   ?person foaf:name ?name .
}
ORDER BY (?name)

Result:
--------
-----------------------
| name                |
=======================
| "Elien Paret"       |
| "Johny Paret"       |
| "Pieter Callewaert" |
-----------------------
```

SPARQL query 1: Simple SELECT query

`CONSTRUCT` allows reformulating bound variables into any kind of RDF graph, this as long as each triple is valid (e.g. no Literals used in the subject or predicate position, all triples are of the form *"<subject> <predicate> <object>"*). This query form allows an easy and powerful way to transform data from one RDF graph or OWL ontology into another. Graphs returned from `CONSTRUCT` queries can be added to RDF repositories or combined with other RDF graphs. SPARQL query 2 shows a construct query which returns an RDF Graph containing information about the "`foaf:name`" of the "`foaf:Person`" objects.

```
SPARQL Query:
--------------
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
CONSTRUCT
{
   ?person foaf:name ?name .
}
WHERE
{
   ?person rdf:type foaf:Person .
   ?person foaf:name ?name .
}

Result:
-------
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf:    <http://xmlns.com/foaf/0.1/> .
```

```
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

[]    foaf:name "Pieter Callewaert" .

[]    foaf:name "Johny Paret" .

<http://wise.vub.ac.be/Elien#me>
      foaf:name "Elien Paret" .
```

SPARQL query 2: Simple CONSTRUCT query

`ASK` is used to know whether a particular graph, a particular binding exists, it responds with a Boolean result. This allows clients to ask endpoints for information without having to submit a potentially-expensive `SELECT` or `CONSTRUCT` query.[2] SPARQL query 3 shows an ask query which returns whether there exists a “`foaf:Person`” with as name ‘Elien Paret’.

```
SPARQL Query:
--------------
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
ASK
WHERE
{
   ?person foaf:name 'Elien Paret' .
}

Result:
-------
True
```

SPARQL query 3: Simple ASK query

`DESCRIBE` returns an RDF graph determined solely by the processor with limited query input from a client. `DESCRIBE` is an interesting case since the client does not need to be intimately familiar with how the data is structured. The endpoint ultimately decides what RDF data is returned to the client. `DESCRIBE` can be useful for building foundational information when data source awareness is not present. It is not used as heavily as the other three forms.[2] SPARQL query 4 shows a describe query that describes everything that the RDF information tells about the “`foaf:Person`” with name ‘`Johny Paret`’.[2]

```
SPARQL Query:
--------------
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
DESCRIBE ?person
WHERE
{
   ?person rdf:type foaf:Person .
   ?person foaf:name 'Johny Paret' .
}

Result:
--------
```

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf:    <http://xmlns.com/foaf/0.1/> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

[]    rdf:type foaf:Person ;
      rdfs:seeAlso <http://wise.vub.ac.be/members/johny> ;
      foaf:mbox_sha1sum "72b9db61f552b422e8d9bac106e063f8ccfea415" ;
      foaf:name "Johny Paret" .
```

SPARQL query 4: Simple DESCRIBE query

There are a couple of modifiers that can be used in a query: `REDUCED`, `DISTINCT`, `ORDER BY`, `OFFSET`, `LIMIT`, `FILTER`, `OPTIONAL`, and `UNION`. These will not be explained further. For more information see reference [2] or [6].

2.7 Semantic Web Frameworks

Semantic Web frameworks are used for many different purposes, including database translation and integration, domain knowledge modeling, validation, analysis, and even simply the storage and retrieval of information. There exist many different Semantic Web Frameworks: Sesame[15], CubicWeb[16], Jena[17] … Regardless of their purpose, most frameworks provide the ability to create and manipulate a knowledgebase. The frameworks are composed of a set of tools, including a RDF store (often referred to as a triple store or graph store), an access API or query processor, and a reasoning engine (or reasoner). Each of these components play a critical role in providing the storage and retrieval of RDF data, as well as the interpretation of the semantics of OWL ontologies and instance data.[2]

The following section explains Jena and µJena, the two Semantic Web frameworks that SCOUT uses to store and access the encountered RDF information.

2.7.1 Jena and µJena

Jena is an open source Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS, OWL and SPARQL. It also includes a rule-based inference engine. The framework has the following features:

- a RDF API,
- reading and writing RDF in RDF/XML, N3, Turtle and N-Triple,
- an OWL API,
- in-memory and persistent storage,
- SPARQL query engine.[7]

[15] http://www.openrdf.org/

[16] http://www.cubicweb.org/

[17] http://jena.sourceforge.net/

µJena (Micro Jena) is a lightweight, reduced ported version of the Jena intended for mobile devices. µJena makes Semantic-Web Framework under Java Micro Edition (J2ME). µJena enables the development of semantic-web applications for mobile devices.

2.8 Caching

The concept of a cache is used in this thesis to cache RDF information. This section will explain the basic concepts of caching. At first a definition of what a cache is, is given, followed by what type of data can be store in a cache. The third section explains how a caching mechanism works. The last two topics, eviction strategy and write strategy, describe the most important issues of a cache.

2.8.1 What

A cache is a component that transparently stores data such that future requests can be served faster [8]. Serving request faster can mean improve the response-time, reduce the number of times accessing the original data over the network, reduce disk accesses …

2.8.2 For what data

There exist several types of caches CPU cache, disk cache, web cache, database cache … Depending on the type of cache and the specified purpose, the content of the cache can be slightly different. A cache can contain objects, a row of a database table, a part of a database table, query results, a web page, and a record of a disk …

2.8.3 How

To explain how a cache works, the example of a web cache is used. The other types of caches (e.g., disk cache) work all in a similar way, only the type of data stored in the cache will be different.

A web cache caches web pages to increase the response time by reducing the number of accesses to the server. Whenever the client requests a web page, the system looks in the web cache before actually downloading the webpage. When the system finds the web page in the cache, this is called a cache hit; the system will get the requested from the cache, and not from the server, and shows it to the client. In this case, it is thus not necessary to access the server. When the page is not found in the cache, called a cache miss, the system ask the requested page to the server and show it to the client.

The explanation above is a very simplistic explanation of how a cache works, with every cache certain issues must be taken into consideration. Firstly one must decide when the data in the cache will be update with data from the original data in order to avoid stale data in the cache. Therefore answering the following question is necessary: is stale data allowed, in what time span should the cached data be update and do we have stale data … Secondly, a strategy, a so-called eviction strategy, which decides what happens when the cache is full must be applied (see the following section for more information). Thirdly, one should consider what write strategy is adopted, in other words, what should happen when the client, accessing the cache data, changes the cache, is the client allowed to change the data … The last section describes the concept of a write strategy.

2.8.3.1 Eviction strategy

An eviction strategy (also called a replacement strategy) is a strategy that will decide what element(s) of the cache will be evicted when it's full. These strategies mostly use frequency and / or recentness to decide what element(s) to evict. The moment at which the replacement strategy decides to evict a page usually happens when a page is requested but not found in the cache.

The eviction strategy Least Recently Used (LRU)[18] is a popular strategy that uses receny to decide what elements to evict. Suppose we have a web cache that uses LRU as eviction strategy. The web cache will always contain the page were used the most recently. When a page is requested and found in the cache, LRU will mark that page as recently used and the web page will be shown to the client. If however the page is not found in the cache, LRU will perform the following steps. First evict the page that was least recently used, then download the new page, add it to the cache and mark it as recently used. Finally show the page to the client.

Some of other well know replacement strategies are First In First Out (FIFO), Last In First Out (LIFO), Least Frequently Used (LFU) and Most Recently Used (MRU).

2.8.3.2 Write strategy

A write strategy is a strategy that will decide what needs to happen when the client changes data from the cache. The two most extreme case are a write trough cache and a write back cache. The first, a write through cache, updates the original data immediately when the client changes the cached data. The original data is thus always the most recent version, is always updated. While a write back cache only updates the original data when the data gets evicted from the cache. The original data is not always last version; it might be out of data, it might be stale. The benefit of a write through cache is that the original data is never stale, contrary to a write back cache where the data might be stale. The drawback of a write through cache is that the original data can get updated quite often, while with a write back the original data is less often accessed, several update actions of the cached data can be combined in one update action of the original data.

Off course these are only the two extreme cases, there exist several intermediate solutions. Which solution is the most appropriate one, depends on the type of cache and the purpose of it.

[18] http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used

Chapter 3 Related Work

This chapter describes the work related to topics discussed in this thesis. The related work is structured into three parts. The first part describes the work related to the SCOUT framework. The second part describes existing work on caching mechanisms, either related or unrelated to Semantic Web technology. The final part discusses approaches that construct and employ summary information on RDF datasets in order to speed up data-access and / or query resolving.

3.1 The SCOUT framework

The SCOUT framework abstracts the user's environment into a set of distinct physical entities, where each entity can have a so-called "Web presence". This Web presence provides information or services associated to the physical entity, in the form of a Website, Web service, online RDF metadata, etc. Whenever the user is nearby a physical entity, the SCOUT framework, running on the user's device, tries to find a reference to the Web presence of that entity. This reference can be obtained by reading an RFID tag nearby the entity. This RFID tag contains a URL pointing to the entity's Web presence, or by accessing an online directory, which provides the SCOUT framework with Web presence references of nearby entities, based on the user's current GPS position. Based on the metadata of physical entities nearby the user, combined with the user's own profile information, the SCOUT framework provides applications with an integrated view of the user's environment called the Environment Model.

3.1.1 Linking physical objects to online resources

Several approaches (both commercial and academic) exist which focus on the linking of physical objects to digital information. The notion of linking physical entities to associated online information was first introduced by the HP Cooltown project [9][10][11], in order to integrate the physical object with the virtual object. They were also the first to introduce the term "Web presence", which denotes online information associated with an entity.

Touchatag[19] from Alcatal-Lucent Ventures is a commercial initiative where RFID technology is used to connect physical objects to online applications. In other words, online application actions can be triggered when a touchatag is read. These actions can range from opening a browser window with the Web address given by the tag, to controlling an online music player, where a set of tags are assigned a certain playback action (e.g., play, stop, open music file). In an online community Website, users can create and share their applications with other users, and configure tags to be linked to online application actions.

[19] http://www.touchatag.com

In [12], an open lookup framework is presented where objects tagged with RFID tags are linked to resources, allowing applications to retrieve information or services related to tagged objects. A resource represents information (e.g., Website) or a service (e.g., Web service) related to a tagged object, while a resource description keeps information on such a resource (e.g., id, URL, title, description, …) and points to the corresponding resource via the URL attribute. Resource descriptions are contained in so-called resource repositories, which can be setup by the manufacturer himself, e.g., representing official product information, or third parties, e.g., representing customer reviews. Several methods are available for applications to locate useful resource repositories: the object's RFID tag-id can be transformed by a resolver service into a resource repository address; a search-service can be used, which crawls known resource repositories and indexes the available resources; and resource repositories can be registered to so-called resource directories. This approach is decentralized, as it allows separate resource repositories to be setup by anyone for any type of object.

The functionality provided by these two latter approaches corresponds to the functionality provided by SCOUT's Detection Layer: i.e., they concern the retrieval of online resources associated with physical entities in the user's vicinity. However, instead of simply providing applications with these resources, the SCOUT framework additionally constructs and maintains a conceptual and integrated view over the information provided by these online resources.

3.1.2 Storage of location- (and context-) specific information

Many approaches which focus on the location-specific retrieval of information employ a centralized Information System (IS), which stores and maintains all location-specific information (e.g., [13] [14][15][16]).

For instance, in [14] and[16] so-called "spatial graffiti" or "digital post-its" allow users to provide loosely structured information on a certain entity, place or region. Based on the user's current location, the system is able to provide relevant post-its to the user. More advanced context-matching methods are provided as well to avoid overloading the user with information: e.g., by matching associated keywords to the user's profile, or by employing the user's own history and the history of similar users (e.g., same age, country of origin) to determine the popularity of certain graffiti [16]. Such graffiti can be directed at certain persons [14] and[16] and can also have a limited lifespan.

Although the ease of input can lead to a huge amount of digital post-its being available, this data is too unstructured for our purposes, as SCOUT focuses on integrating the information extracted from the user's physical environment. However, [17] also proposes (as future work) to extract ontologies based on the keywords associated with the post-its, so related post-its can be explicitly linked together.

In [13], a location-aware Web system is proposed where users can easily enter and update information on a certain place (e.g., time of events, organizational structure, description …). The paper states that solutions like spatial graffiti are too loosely structured, and focuses on keeping a careful balance between ease of input and structuring of information, to guarantee that the information is sufficiently usable and navigable while still making it easy for users to contribute information. Content providers can define a place by drawing a rectangular box on a predefined map; subsequently, data can be associated with the pinpointed place.

Physical hypermedia applications (e.g., [15]) are hypermedia systems where the nodes represent real-life objects, and links between them can be either digital (i.e., hyperlinks) or physical (i.e., representing a route between the objects). In these systems, physical objects are augmented with digital information: when a user stands nearby such an object, he can access related digital information. Digital links to other nodes can be followed in the normal (Web-like) way, while physical links involve having the user cross the distance to the linked physical object. For the latter, instructions (e.g., on a map) are provided on how to get there. In [15], an OOHDM extension is proposed, which provides support for physical hypermedia in the OOH design method. This extension supports different location models (symbolic / geometric / ...) in order to link physical objects to the corresponding digital information.

In all these approaches, a single system is responsible for storing and providing access to location-specific information on a certain region. This system thus presents a bottleneck and possible point-of-failure, as all requests for location-specific information have to pass through this system. Furthermore, these systems mostly require pre-defined models of the area, which is not very flexible, and unsuitable for dealing with non-stationary entities. Finally, uploading the necessary (structured) information to a single, closed system means that the uploaded content is not available to third-party services.

In contrast, the SCOUT framework re-uses existing online resources, and integrates them into the conceptual view provided to the applications on top of SCOUT. Therefore, content providers are free to upload their data to any WWW server, thus keeping control over their own data and allowing it to be re-used by other applications or services. Furthermore, as SCOUT runs locally on a user's device, there is no single point-of-failure that can block information retrieval for an entire region. Finally, the Detection Layer of SCOUT supports a variety of detection techniques (e.g., RFID, GPS), allowing for the detection of dynamic entities (e.g., moving persons) and minimizing the need for predefined location models.

3.1.3 Integrated view over distributed data sources

In contrast to the approaches in the previous subsection, which store all location- (and context-) specific information in one place, approaches such as [17] and[18] provide a central service which acts as an integrated view over a set of distributed data sources.

For example, in [17] a middleware is proposed which matches (local) schemas of heterogeneous (context) data sources to a set of global schemas, in order to provide applications with a unified abstract view of the context sources. It is argued that because the schema matching occurs in a mobile environment, existing schema matching approaches cannot be reused directly. A name-based matcher (using several criteria such as equality, stemming synonyms, etc) is employed for matching individual attributes of the local schemas to the attributes of the global schemas. The middle ware they present is a complete solution, as it provides a SQL-based query interface over the global schemas. In this approach, given queries are distributed across the registered context sources, which are responsible for transforming them to fit the local data schema and subsequently returning the query results. Finally, these results are combined by the middleware and returned to the application.

A Context Information Service (CIS) is presented in [18], which represents a virtual database over registered context sources. The schema of this database corresponds to the data types available from the different sources, and a Context Synthesizer is responsible for distributing the query across the registered sources and integrating the results. In addition to specifying a query, a client can also provide meta-attributes which denote the required attributes of the results (e.g., accuracy, reliability); that way, the most suitable context source can be chosen in case some of them overlap.

These approaches provide an external service which handles queries over data sources, thus offloading a lot of work from the mobile devices themselves [18]. However, they suffer the same drawback as the systems discussed in the previous subsection: i.e., they are less scalable and flexible, as every query needs to pass through this service and each data source needs to be registered with it. Furthermore, they impose severe restrictions on data sources, requiring them to have a specific interface and be able to execute (parts of) queries; this rules out sources which simply represent files containing metadata. Finally, with more advanced mobile devices being released every year, the need for so-called “thin” clients (i.e., lightweight client-side applications) is constantly being reduced.

3.1.4 Other decentralized approaches

An approach that also focuses on the decentralized, context-specific retrieval of data is Context-ADDICT [18]. In this approach, a Domain Ontology is constructed modeling the main concepts in the application domain; also, a Context Dimension Tree is constructed which models the user’s context. Subsequently, relations between the two models are constructed in order to capture the part of the Domain Ontology that is relevant to the user. For each encountered data source, a Semantic Extractor module is employed to extract the ontology used in the data source, which is matched and integrated into the Domain Ontology, leading to a Merged Schema. By using the original relations between the Domain Ontology and the CDT, it can be deduced from this Merged Schema what parts of the encountered sources are relevant to the user. This part is called the Local Schema. Each given query will be split up into two parts: a retrieval part, which will be distributed across the relevant sources (as indicated by the Local Schema), where the sources are responsible for transforming the query part to the correct format and executing it, and a reasoning part, which will be executed on the results of the retrieval part of the query. This paper thus focuses on schema-based filtering, and does not provide an event-based way of obtaining environment information. Moreover, as was the case in the previous subsection, this approach requires data sources to support a specific interface and to be able to execute (parts of) queries, ruling out data sources simply representing a metadata file.

As mentioned before, the notion of linking physical entities to associated online information was first introduced by the HP Cooltown project [9] [10] [11], in order to integrate the physical with the virtual world. In addition, it also introduced the concept of positional relations, which denote that users are nearby certain physical entities. In [9], the generation of a Web presence (i.e., Website) is made dependent on the context of the client, including the current set of positional relations he is currently involved in. [10] focuses on making Web presences for places, where either a place's Web presence can be accessed to navigate to the Web presences of the entities contained in that place, or a place manager component is used to resolve given entity identifiers to Web presence URLs. Furthermore, [10] and [11] present a Web transfer model, which comprises the direct / indirect transfer and retrieval of content from "Web present" objects (i.e., objects having a Web presence; for instance, transferring pictures from a PDA to a projector).

SCOUT is also based on the linking of physical objects to digital information, and re-uses the concepts of Web presences and positional relations. However, in contrast to the Cooltown project, SCOUT specifically focuses on context-sensitive information retrieval: this is best illustrated by the Environment Model, which provides a conceptual and integrated view on all of the information available from the user's physical environment, and the Filtering Service, which allows push-based information retrieval based on complex semantic conditions.

3.2 Caching mechanisms

This thesis investigates how the Environment Model (i.e., encountered data sources) within the SCOUT framework can be constructed and managed more efficiently than before, when it was a fully materialized view; in other words, all of the data (i.e., encountered data sources) is kept locally. A part of this investigation is to develop a caching mechanism for the encountered data sources which contain RDF information.

3.2.1 Caching query results

[19] and [20] both caches query or sub-query results.

The proposed cache mechanism of [19] is intended for a peer-to-peer environment where each peer has its own cache. The entries in the cache are actual (sub) query result or references to peers that offer the query result. In their approach they also propose several strategies to determine what queries must be cached (e.g., store the result if it has been more requested than a particular threshold, only store the query result if calculating it is more expensive than storing it, …). They do not compare the different strategies they propose, their approach uses the strategy where a query result is stored when it has been requested more than a particular threshold. However they mention that comparing the different proposed strategies is future work.

In [20] they also adopt the approach to cache (sub) query result. However a cached query result may not only solve the original query but also a similar query. In their approach they transform, adapt, the cached query result into the new query results. Before actually transforming a cached query, they compare the cost of modifying the cached query result to the cost of actually calculating the new query result.

In this thesis the approach of caching query results is not followed. However we do follow the approach of [19] where each peer, in our case a client running the SCOUT framework, has its own cache. The approach to invent different strategies to determine what will be cached is adopted as well. We do however go further than just inventing them; we actually evaluate the different approaches, discuss them in detail by giving the benefits and drawbacks.

3.2.2 Caching data client side

In contrast to the approach followed in the previous section, where query results are cached, this section discusses the approach proposed by [21] in which Semantic Data is cached in a on the client-side of a client-server database system.

Their approach proposes to have a client-side cache that is composed out of so-called semantic regions, which contains multiple tuples. The size of such a semantic region is not fixed, it can vary along with the number of tuples that is contained in it. Each region has a constraint formula describing its content, the number of tuples that satisfy the constraint, a pointer to the linked list containing the tuples and some additional information that is used by the eviction strategy. What is so special about this semantic cache is that the regions can intersect as Figure 5 shows, Q1, Q2 and Q3 are all so-called regions.

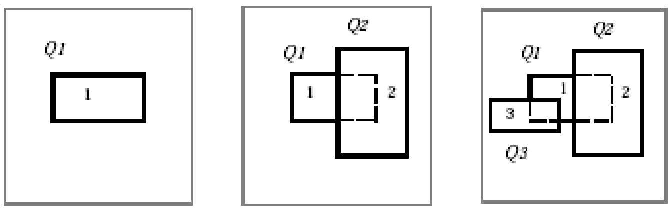


Figure 5: Example content semantic data cache

Our second caching mechanism, which caches triples that use a specific predicate as one cache entry, resembles to the approach discussed in [21]. We also keep the granularity on the level of tuples (i.e., RDF triples), our constraint formula is a formula stating that a specific predicate should be used, the size of our cache entries, regions is dynamic (i.e., it varies along with the number of triples contained in it). However our cache entries, regions do not share tuples since one triple can only use one specific predicate.

3.2.3 Web caching

In [22] and [23] a caching algorithm is proposed for caching web pages, web objects. [22] proposes the Resource Based Caching algorithm, an algorithm which is placed on a Web Server, while [23] propose a client side caching mechanism which stores RDF resources within Haystack, a Semantic Web Browser.

[22] proposes the resource based caching algorithm (RBC), which is situated in the domain of web servers. In [22] it is stated that there is (will be) a need to not only cache text and images but also hyper-media objects and that caching algorithms should be adopted in order to allow caching of these multiple data types and their heterogeneous requirements. The RBC algorithm characterizes each (hyper-media) object by its caching gain and resource requirement (e.g., necessary size and bandwidth), dynamically selects the granularity of the object to be cached that minimally uses the limited cache research (i.e., bandwidth or space), and if required, replaces the cached objects based on their cache resource usages and cache gain. This cache gain depends on the metric used to measure the cache performance (e.g., total bandwidth saved, bytes transferred per second, …).

The developed caching mechanism within the SCOUT framework will not cache hyper-media objects, it will only cache RDF information, RDF triples. Currently the SCOUT framework decides which items to replace based on their cache resources usage, however in future work the idea of replacing items based on their cache resource usage and cache gain will be adopted.

In [23] a Semantic Web browser, called Haystack is presented. This Semantic Web browser has as benefit that it can use the RDF metadata to allow the user to gain direct access to the underlying information and control how it is presented. This is in contrast to the current browsers that display html pages of which the user cannot or can only partially determine how the information should be displayed. Haystack automatically locates metadata and creates point-and-click interfaces from a combination of relevant information, ontological specifications and presentation knowledge, all described in RDF and retrieved dynamically from the Semantic Web.

The RDF resources that are visited by the user with Haystack are cached for network efficiency reasons. They use an ordinary local RDF store to cache RDF data from other sources. They just add the consulted RDF information to the cache. When requesting information, the request is first resolved using the data in the cache, unresolved URI's mentioned in the request are fetched and the metadata is added to the store. Our approach resembles to this approach as we will also (in one of our caching mechanism) cache entire RDF resources locally.

3.3 Indexes

This thesis investigates how the Environment Model (i.e., encountered data sources) within the SCOUT framework can be constructed and managed more efficiently than before, when it was a fully materialized view; in other words, all of the data (i.e., encountered data sources) is kept locally. A part of this investigation was to know which sources contain which information, therefore a source index model was created. [24], [19] and [25] all present indexes to determine relevant source for a specific query. [24] stores information about the predicate and objects but also statistical information like number of triples to determine which source is relevant. While [19] uses subject, predicate and object to determine the relevant source. [25] uses so-called join-indexes spread over different repositories.

In [24] DARQ, a query engine for federated SPARQL queries, is presented. It provides transparent query access to multiple, distributed endpoints as if querying a single RDF graph. Such an endpoint is thus capable of handling a SPARQL query independently, and returns the results. To indicate the capabilities of an endpoint, a service description, comparable to our summary information of an encountered source, is used.

This service description is a declarative description of the data that the endpoint offers based on predicates and described in RDF (see Example 1). Based on these service descriptions, the query planner finds the relevant sources and it decomposes the query into sub-queries, which can be executed by one relevant source. The results of all these sub-queries can be combined and form the result of the original query. The endpoints are asked to execute an optimized version of these sub-queries.

```
[ ]   a  sd:Service ;
      sd:capability [ sd:predicate foaf:name;
                       sd:objectFilter "REGEX(?object , "^[A-R];
                       sd:triples 51 ];
      sd:capability [ sd:predicate foaf:mbox;
                       sd:triples 51 ];
      sd:capability [ sd:predicate foaf:weblog;
                       sd:triples 10 ];
      sd:totalTriples "112";
      sd:url "EndpointURL";
      sd:requiredBindings  [ sd:objectBinding foaf:name ];
      sd:requiredBindings  [ sd:objectBinding foaf:mbox ].
```

Example 1: DARQ, service description

Our approach will summarize less information, only the used predicates (and used domains) are stored, while in [24] other information (e.g., the range of a predicate, the number of triples using the predicate, the total amount of triples …) is stored as well. From our test and evaluation section we came to the conclusion that maintaining more information (i.e., predicate and domain) is less efficient than just storing predicate information. Investigating other, more complex, summary information is considered to be future work.

In [24] it is required that every physical entity has such an endpoint, capable of handling SPARQL queries independently and returning the results. However we do not take this assumption, our entities will not always have an endpoint capable of handling SPARQL queries, our entities have minimal an online RDF source.

In paper [19] the caching of intermediate results in Distributed Hash Tables (DHT), a mechanism used to store RDF triples, is discussed. Project using these DHT's often not only distribute the storage but the query load as well in order to enhance the scalability. These systems index the information by subject, predicate and object to determine which peer is responsible for the subject, predicate or object.

Our approach does not follow the idea to distribute the query load to the peer that is offering the information. Currently, the information is stored in a distributed manner, but the query is always resolved by the main entity. It is considered future work to provide a mechanism that will solve queries either locally or remotely, by means of a query service.

In a DHT RDF information is indexed by subject, predicate and object while our mechanism only indexes the information by predicate (and by domain, subject).

In [25] the extension of Sesame[20], an open source framework for storage, inferencing and querying of RDF data, is propose to allow querying of distributed RDF repositories. Currently the Sesame framework and other similar frameworks like Jena[21] do not allow querying distributed

[20] http://www.openrdf.org/

[21] http://jena.sourceforge.net/

RDF repositories. The approach proposed in [25] will firstly analyze the query to determine which repositories contain relevant information and what part of the query will be answered by which repository. Secondly the appropriate queries are send to the relevant repositories, their information is merged and returned. By delegating the query (or parts of the query) the query execution process is optimized. This approach again assumes that the repositories have querying capabilities is contrast to our assumption, where an encountered entity does not always has querying capabilities.

In order to determine which repositories contain which information, a specific index structure, a so-called join index which contains the result of a join over a specific query, is used. The proposed index structure contains information about a single property and about paths or combinations of properties (e.g., A foaf:knows B foaf:mbox C). These paths can then be used to determine which repository contains the answer for (a part of) a path.

Chapter 4 Architectural overview

Previously the Environment Model has been stored as a fully materialized view; in other words, all of the data (i.e., encountered data sources) was kept locally. This thesis investigated how the Environment Model can be constructed and managed more efficiently.

The first step of the proposed approach is, storing source index information on encountered data sources as a Source Index Model, to determine which sources contain relevant information for a given query. The purpose is to avoid having to include all encountered sources when answering a query and thus accelerate the process of resolving a query.

Secondly, we have developed several caching strategies where some data from encountered sources is kept locally, to avoid having to download a relevant source (identified as relevant by the Source Index Model from above) every time it is needed to solve a query issued to the Environment Model.

The following section explains how these two mechanisms work together in the SCOUT framework by explaining how they are used when the Environment Model is accessed and what happens when a new source is encountered. The two following chapters explain each mechanism in detail, the Source Index Model in Chapter 5 and the cache strategy in Chapter 6.

4.1 Accessing the Environment Model

The SCOUT framework, currently being developed at the WISE lab, supports the development of context- and environment-aware mobile applications, by providing a conceptual and integrated view on the environment called the Environment Model. This Model comprises metadata on physical entities found nearby the user and the user's own profile information, and thus allows applications to become aware of (and responsive to) the user's physical environment and context.

This Environment Model consists of three parts (see Figure 6): the comprised metadata, the Entity Model, which contains the user's own profile information and the Relation Model, a model that contains which entities where previously or currently nearby our user.

When a mobile application is accessing the Environment Model, he is actually accessing the Entity Model, the encountered data sources and the Relation Model. Accessing the encountered sources means retrieving the relevant information from the cache. To determine what information is relevant for the mobile application, the cache uses the Source Index Model, which is capable of determining the relevant sources for a specific query. After determining the relevant information, the cache retrieves all the relevant information either by accessing its local copy of the relevant information or by downloading the information.

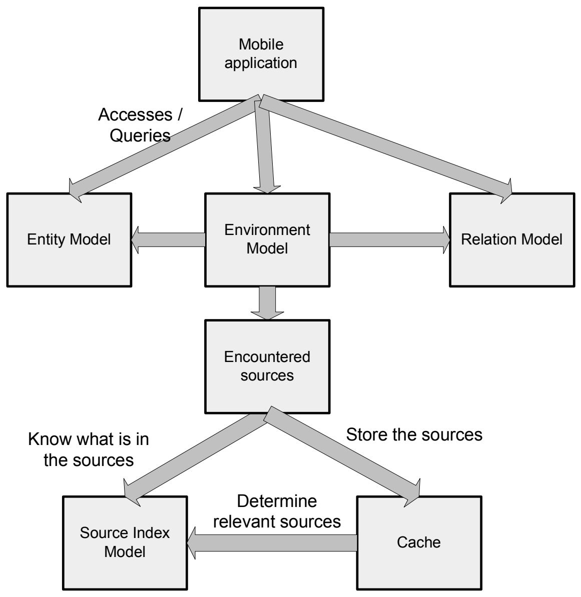


Figure 6: Accessing the Environment Model

4.2 Encountering new sources

In a mobile environment, the context and environment changes frequently. The SCOUT framework supports this by updating the models capturing the context and environment immediately when new sources are encountered. The Relation Model and Environment Model are notified when a new data source is encountered since they maintain the context- and environment of the user.

When updating the Environment Model, the Source Index Model is updated as well since otherwise when consulting the Source Index Model for relevant data of a given query the newly encountered source will not have been considered since the model does not know it exists.

When updating the Environment Model, the cache needs to be update as well because it must have the opportunity to decide whether or not he will actually store the newly encountered source. The figure below illustrates the interactions that occur when a new source is encountered.

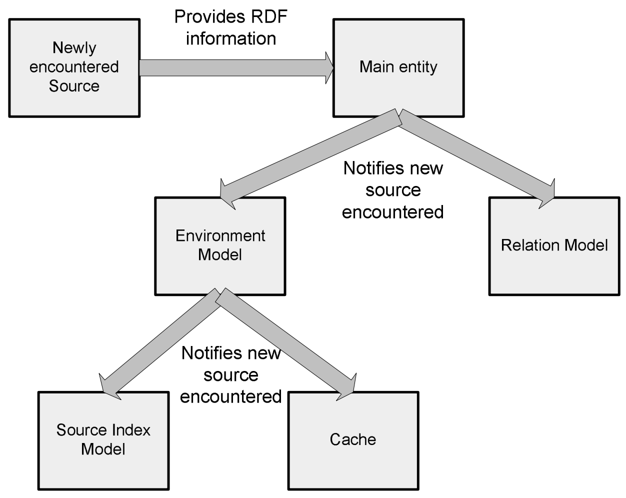


Figure 7: Encountering a new source

Chapter 5 Source Index Model

The purpose of this thesis is to investigate how the Environment Model can be constructed and managed more efficiently. A first step in the solution is to maintain a Source Index Model that contains source index information telling us what kind of information is used in a specific source. This model is then used to determine which of the encountered data sources are relevant for resolving a query. This allows us to no longer consider all the encountered sources when a query is posed to the Environment Model, but only consider the relevant sources.

This chapter explains the Source Index Model by firstly explaining what information can be stored in it, from this exposition two possible variants are chosen called Source Index Strategy1 and Source Index Strategy 2. Both strategies store their Source Index Model as an RDF Graph, section 2 explains the RDF Schema that is developed to represent the Source Index Model. Section 3 and 4 explain what happens to the Source Index Model when a new source is encountered and when it is accessed to determine the relevant sources, and this for both variants. The fifth and last section provides the details of the query analyzing process, this processes extracts from a given query the relevant information to pose to the Source Index Model.

Before elaborating on the concept of the Source Index Model, we must point out that our approach is not yet complete. Currently we have abstracted from the fact that the encountered sources can change, in the current approach, we assume that once the sources are encountered, they never change and thus their source index information does not need to be updated. It is considered future work to further develop the mechanism in such a way that it supports change in the encountered data sources (see Chapter 9).

5.1 What information to store?

This first section investigates what information can be summarized from encountered data sources in the Source Index Model. The difficulties with these encountered sources are that they can be encountered at any time and they do not have fixed data schema. Especially the lack of a fixed data schema is a complicating factor. One source can contain information about movies, another information about persons, yet another information about restaurants, etc. This part of this thesis focuses on extracting and storing certain summarizing data from a source (e.g., predicates it contains, types of domains used, range of the predicates, namespaces, number of predicates), and store it in the so called Source Index Model. The purpose of doing so is to be able to determine which sources are relevant for a particular query, without the need to exhaustively query all the sources. There are several possible variants for a Source Index Model, depending on which information is stored. In this thesis we investigate two possibilities which are described in the following paragraphs.

As RDF is a predicate-oriented formalism, a natural choice is to store these predicates, possibly along with some additional information (such as domain and range). Compare with an Object-Oriented system, where it is natural to maintain summary information about the objects, and possible their methods, variables, etc. Two variants of such predicate-oriented Source Index Model are explored here: one that stores information only about the used predicates, and a second that stores information about used predicates and their corresponding domains. The benefit of the first option is that it results in a smaller source index model. Secondly, it causes fewer overhead to create and maintain, since it is not necessary to determine, for each predicate, the domain. The benefit of the second option is that is increases selectivity. To validate these conjectures, we will compare these two Source Index Model strategies using a series of experiments; details are given in Chapter 8.

5.2 RDF Schema for the Source Index Model

Both our source index strategies store the model as an RDF Graph; this allows us to access the model in the same way as the encountered data sources are accessed. To be able to store the Source Index Model properties and classes of existing RDF Schemas (i.e., rdf[22] and rdfs[23]) are used. This was however not enough, we had to developed a new RDF Schema, shown in RDF data 6, to fit our needs. This schema defines one class, “`meta:Source`”, which represents an encountered data source containing RDF information, and four properties. The first property “`meta:hasDomain`” states that a particular property (i.e., predicate) is used with a particular domain (i.e., the type of the subject). The other three properties, denoted by “`meta:presentIn`”, are used to state that something (i.e., statement, domain or property) is used within a specific source.

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY meta "http://wilma.vub.ac.be/~eparet/metafile008_2.rdfs#">
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
]>

<rdf:RDF
   xml:base="&meta;"
   xmlns:rdf="&rdf;"
   xmlns:rdfs="&rdfs;"
   xmlns:meta="&meta;"
   >

<!-- Defining classes  -->
<rdfs:Class rdf:about="&meta;Source">
   <rdfs:comment>The class of file / Web presences containing RDF information</rdfs:comment>
</rdfs:Class>
```

[22] http://www.w3.org/1999/02/22-rdf-syntax-ns

[23] http://www.w3.org/2000/01/rdf-schema

```
<!-- Defining properties -->
<rdf:Property rdf:about="&meta;hasDomain">
   <rdfs:domain rdf:resource="&rdf;Property" />
   <rdfs:range rdf:resource="&rdfs;Class" />
   <rdfs:comment>Relation saying that the Property has the Class as Domain.</rdfs:comment>
</rdf:Property>

<rdf:Property rdf:about="&meta;presentIn">
   <rdfs:domain rdf:resource="&rdfs;Statement" />
   <rdfs:range rdf:resource="&meta;Source" />
   <rdfs:comment>Relation saying that the Statement (Property hasDomain Class) is present in the
Source</rdfs:comment>
</rdf:Property>

<rdf:Property rdf:about="&meta;presentIn">
   <rdfs:domain rdf:resource="&rdfs;Class" />
   <rdfs:range rdf:resource="&meta;Source" />
   <rdfs:comment>Relation saying that the Class is present in the Source.</rdfs:comment>
</rdf:Property>

<rdf:Property rdf:about="&meta;presentIn">
   <rdfs:domain rdf:resource="&rdf;Property" />
   <rdfs:range rdf:resource="&meta;Source" />
   <rdfs:comment>Relation saying that the Property is present in the Source.</rdfs:comment>
</rdf:Property>

</rdf:RDF>
```

RDF data 6: RDF Schema used to store the source index models

Since our first source index strategy maintains which source uses what predicate with which domain, we need to be able to express that the predicate and corresponding domain are used within a particular source. It is not sufficient to express that a particular domain is used within a source, and the predicate is used within the source; we aim to express the combination of domain - predicate used in a particular source. In other words, we want to express that the statement "`<predicate> meta:hasDomain <domainOfSubject>`" occurs in a particular source. To do so, we use the RDF reification process.

The following example is used to motivate that it is not sufficient to state that a predicate and a domain are used in a source, without explicitly stating the combination of both. Consider two sources called source A and source B as shown in RDF data 7. RDF data 8 shows the source index information of source A and source B. By examining the source index information, we could conclude that both "`dc:title`" and "`foaf:Person`" are used in sourceA and that "`dc:title`" has "`foaf:Person`" as domain. We could thus (mistakenly) conclude that the combination "`dc:title`" and "`foaf:Person`" is present in source A. However, this is not the case, as the statement expressing that "`dc:title`" has as domain "`foaf:Person`" actually originates from source B. The problem is thus that domain information about a certain predicate present in one source, might not apply to the same predicate in another source. If we do not explicitly store the domain

– predicate information in our Source Index Model, we are no longer able to determine which domain is used in which source.

```
Source A:

<person1> a foaf:Person.
<person1> foaf:name 'Elien Paret' .
<track1> a mo:Track .
<track1> dc:title: 'Envoi' .

Source B:

<person2> a foaf:Person .
<person2> dc:title 'Mr' .
```

RDF data 7: Source A and B

```
Summary information of Source A and B:

foaf:Person meta:presentIn sourceA .
foaf:name meta:presentIn sourceA.
foaf:name meta:hasDomain foaf:Person .

mo:Track meta:presentIn sourceA .
dc:title meta:presentIn sourceA .
dc:title meta:hasDomain mo:Track .

foaf:Person meta:presentIn sourceB .
dc:title meta:presentIn sourceB .
dc:title meta:hasDomain foaf:Person .
```

RDF data 8: Source index model information source A and source B

The RDF Reification process is used to express that a predicate-domain combination occurs in a particular source. In this process the "`<predicate> meta:hasDomain <domain>`" triple is translated into a RDF resource that can be addressed. Traditionally, this RDF resource is a blank node with a random uniquely identifier. However this random uniquely identifier is not sufficient enough for our purpose. Since adding twice the same predicate-domain combination results in two triples that are reified into two separate blank nodes with their own random identifier. We want to avoid duplicating the predicate-domain triple and thus to be able to reuse the node, therefore we create our own unique identifier to give to the reified statements, these all get an URI that starts with "`meta://`" followed by an unique number. RDF data 9 shows an example of a reification of the triple "`<predicate> meta:hasDomain <domain>`".

The rdfs schema as described in RDF data 6 is sufficient to store both variant described in this thesis. More extended versions of the Source Index Model could require extensions of this RDF Schema; this is considered future work.

5.3 Adding source index information of a new source

The Source Index Model contains summarizing meta-data of all encountered data sources in order to determine the relevant sources for a particular query. To keep this model up-to-date, each time a new source is encountered, the Source Index Model needs to be updated with source index information of the newly encountered source.

Extracting the relevant source index information from a source is done by using a specific SPARQL query to the source. This query depends on the particular information that needs to be stored in the Source Index Model. For each variant of the Source Index Model, the query that is used slightly differs. Once the relevant meta information is extracted, it is added to the RDF Graph representing the Source Index Model.

The following subsections show the SPARQL query used to extract the relevant meta information (which we will call source index information from now on) from a particular source, both for Source Index Strategy 1 and Source Index Strategy 2. We also discuss the Source Index Model in more detail for both strategies.

5.3.1 Source Index Strategy1

This first subsection describes how the SPARQL query posed to an encountered source and the Source Index Model look like for the first variant, called Source Index Strategy1, where for each source the used predicates and corresponding domains are maintained.

SPARQL query 5 shows the query posed to encountered sources. In this query "`?c`" represents the domain and "`?p`" represents the predicate; a filter is used to exclude the predicate "`rdf:type`", as it is used in all sources to indicate domain information and thus does not bring any differentiating information. The domain of a subject is not always indicated in a query, in other words the triple "`<subject> rdf:type <domain>`" is not always used in a query. We do not want to oblige that the query must explicitly give a domain for all the used subjects. Secondly, an RDF source does not always mention the domain of a subject himself. Therefore we have chosen to use the domain information in the query if it is given, otherwise only the predicate information is used. This is the reason why we choose to retrieve the domain in via an "`optional`" clause. Using the "`DISTINCT`" operator assures that the domain-predicate pairs are returned without duplicates.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?c ?p
WHERE
{
   ?x ?p ?y .
   OPTIONAL
   {
      ?x rdf:type ?c .
   }
   FILTER (?p != rdf:type )
}
```

SPARQL query 5: constructing source index model (SourceIndexStrategy1)

RDF data 9 is a piece of the source index information of the encountered data source "`http://wilma.vub.ac.be/~eparet/elien_foaf_N.rdf`" (see 11.1 for the actual content of the RDF file). As stated before, the source index information is represented in RDF itself. This piece of source index information tells us that the source uses triples with as property "`foaf:givenname`" and as domain "`foaf:Person`".

```
<http://wilma.vub.ac.be/~eparet/elien_foaf_N.rdf> <rdf:type> <meta:Source> .

<foaf:Person> <rdf:type> <rdfs:Class> .

<foaf:givenname> <rdf:type> <rdf:Property> .
<foaf:givenname> <meta:presentIn> <http://wilma.vub.ac.be/~eparet/elien_foaf_N.rdf> .

<meta://211593053> <rdf:type> <rdf:Statement> .
<meta://211593053> <rdf:object> <foaf:Person> .
<meta://211593053> <rdf:predicate> <meta:hasDomain> .
<meta://211593053> <rdf:subject> <foaf:givenname> .
<meta://211593053> <meta:presentIn> <http://wilma.vub.ac.be/~eparet/elien_foaf_N.rdf> .
…
```

RDF data 9: partial source index information for http://wilma.vub.ac.be/~eparet/elien_foaf_N.rdf (SourceIndexStrategy1)

5.3.2 Source Index Strategy 2

This section describes how the SPARQL query posed to the encountered source and the Source Index Model look like for the second variant of the Source Index Model, which only stores predicates. This variant is called Source Index Strategy 2.

The query posed to the encountered source is slightly different from the one used for Source Index Strategy 1, as the actual domain no longer needs to be captured. The query to extract the relevant source index information, shown in RDF data 10, is thus simpler.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?p
WHERE
{
   ?x ?p ?y .
   FILTER (?p != rdf:type )
}
```

SPARQL query 6: constructing source index model (SourceIndexStrategy2)

The RDF information in RDF data 10 is a piece of the source index information for the encountered data source "`http://wilma.vub.ac.be/~eparet/elien_foaf_N.rdf`". This source index information tells us that the source contains "`foaf:givenname`" as a property (predicate). Note that the source index model for this source index strategy is much smaller than for Source Index Strategy 1.

```
<http://wilma.vub.ac.be/~eparet/elien_foaf_N.rdf> <rdf:type> <meta:Source> .

<foaf:givenname> <rdf:type> <rdf:Property> .
<foaf:givenname> <meta:presentIn> <http://wilma.vub.ac.be/~eparet/elien_foaf_N.rdf> .
…
```

RDF data 10: partial source index information for http://wilma.vub.ac.be/~eparet/elien_foaf_N.rdf (SourceIndexStrategy2)

5.4 Accessing the Source Index Model

The purpose of maintaining a Source Index Model is to be able to determine the relevant source for a query when it is posed to the Environment Model. To do this, Source Index Strategy constructs a specific SPARQL query, called the source index query, which he poses to the Source Index Model.

To construct this Source Index Query, the Source Index Strategy analyses the given query to determine which predicates (and domain) it uses. This process of analyzing the given query is explained in section 5.5.

The result of this Source Index Query contains all the relevant sources need to resolve (part of) the query. Each of these sources contains at least one predicate (and domain) of the query to resolve. Although such a (single) source might not be sufficient by itself to solve the complete query, combined with other the other sources, it might be able to address the complete query. Therefore we need to combine, "`UNION`", all the sources containing relevant information for a part, minimum one "`WHERE`" clause condition, of the original query.

5.4.1 Source Index Strategy 1

To explain the working of Source Index Strategy 1 we will consider an example. SPARQL query 7 is a query that selects all the names of persons known by 'Elien Paret'. The corresponding Source Index Query to determine the relevant sources for SPARQL query 7 is shown in SPARQL query 8.

In the original query, two where patterns are used to express a subject type (numbers 2 and 5). The first where pattern (number 1) uses the "`foaf:name`" predicate, we know from number 2 that the corresponding domain of this subject (?p1) is "`foaf:Person`", we can thus conclude that the "`foaf:name`" – "`foaf:Person`" predicate domain combination occurs. The third pattern uses the "`foaf:knows`" predicate, with the same subject as the first pattern, we can thus deduce the "`foaf:knows`" – "`foaf:Person`" predicate-domain combination. The fourth were pattern uses the “foaf:name” predicate this time with subject ?p2 in the fifth were pattern the domain of this subject is given namely “foaf:Person”, we thus again find the “foaf:name” – “foaf:Person” predicate-domain combination. To solve our original query (SPARQL query 7) we need all the sources that use the combination “foaf:name” – “foaf:Person” and / or “foaf:knows” – “foaf:Person”.

In our Source Index Model, the predicate-domain information is stored as a statement created by the reification process (see RDF data 9 for an example of the Source Index Model). In our Source Index Query we will not specify the name of the reified statement indicating the predicate-domain information since this is not necessary, we simply address it by using a variable (e.g. ?_1) is sufficient and is simpler than calculating the name of the reified statement. We used the reified statement to express “[<predicate> meta:hasDomain <domain>] meta:presentIn <source>”, in terms of the reified statement this is translated into four where patterns shown in RDF data 11.

```
?statement <rdf:subject> <predicate> .
?statement <rdf:predicate> <meta:hasDomain> .
?statement <rdf:object> <domain> .
?statement <meta:presentIn> <source>
```

RDF data 11: Reifeid statement of predicate-domain combination

Our Source Index Query contains twelve where patterns, six for each predicate-domain occurrence. Four of the six where patterns are used to express the predicate-domain combination and two to express that the predicate is a “rdf:Property” and the domain is a “rdfs:Class”. The six patterns of the first predicate-domain occurrence are combined via a “union” to the six patterns of the first predicate-domain occurrence, this expresses the and / or relationship, in other words, we want to have all the sources that use the first predicate-domain combination and / or that use the second predicate-domain combination.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE
{
   ?p1 foaf:name 'Elien Paret' . (1)
   ?p1 a foaf:Person . (2)
   ?p1 foaf:knows ?p2 . (3)
   ?p2 foaf:name ?name . (4)
   ?p2 a foaf:Person . (5)
}
```

SPARQL query 7: Example query posed to the Environment Model

```
SELECT DISTINCT ?source
WHERE
{
   {
      ?_0 <rdf:subject> <foaf:name> .
      ?_0 <rdf:object> <foaf:Person> .
      <foaf:Person> <rdf:type> <rdfs:Class> .
      ?_0 <rdf:predicate> <meta:hasDomain> .
      <foaf:name> <rdf:type> <rdf:Property> .
      ?_0 <meta:presentIn> ?source .
   }
   UNION
   {
      ?_1 <meta:presentIn> ?source .
      ?_1 <rdf:predicate> <meta:hasDomain> .
      ?_1 <rdf:subject> <foaf:knows> .
      foaf:Person> <rdf:type> <rdfs:Class> .
      ?_1 <rdf:object> <foaf:Person> .
      <foaf:knows> <rdf:type> <rdf:Property> .
   }
}
```

SPARQL query 8: Source Index Query (SourceIndexStrategy1)

However, Source Index Strategy 1 can also wrongfully exclude sources that can still yield query results when they are combined (so-called false-negatives). These false-negatives occur when one source indicates the type of a certain resource, while another source uses the same resource as subject of a predicate used in the query, both sources are falsely considered irrelevant. In that case, Source Index Strategy 1 will not detect the latter source as relevant, as it expects the type of a subject resource (i.e., "actual" domains of predicates) to be specified in the same source as they occur as subject of a predicate. However, we expect that the amount of false-negatives is rather small, since the typing of a subject resource and their use with predicates usually occur in the same source. Future work will be investigating how often these false-negatives occur in practice and how they can be avoided.

5.4.2 Source Index Strategy 2

The previous section showed, at hand of an example, Source Index Query for SourceIndexStrategy1. This section shows the Source Index Query for the second variant of the Source Index Model, SourceIndexStrategy2, which only contains information about the used predicate.

The Source Index Query for SourceIndexStrategy2 for the same original query (SPARQL query 7) is shown as SPARQL query 9. As we are now only concerned about relevant predicates (i.e., "`foaf:name`" and / or "`foaf:knows`" in the example), the Source Index Query is simpler. In the Source Index Model, the use of a predicate in a source is summarized in two triples, a first triple defining that the predicate is a property, the second that it is present in a particular source. In our Source Index Query we have thus two where patterns per used predicate from the query, one defining that the predicate is a property and secondly one expressing that the predicate is used in a particular source. Again the where patterns of one predicate are combined via the "`union`" operator which expresses the and / or relationship, since we are interested in the sources that either use the first and / or the second predicate.

Since this strategy uses fewer constraints than SourceIndexStrategy1, it is expected that the number of relevant sources it returns is larger than the number returned by the SourceIndexStrategy1. Several experiments were executed to test which strategy is the most suitable; Chapter 8 describes these experiments and their outcome.

```
SELECT DISTINCT ?source
WHERE
{
   {
      foaf:knows rdf:type rdf:Property .
      foaf:knows meta:presentIn ?source.
   }
   UNION
   {
      foaf:name meta:presentIn ?source .
      foaf:name rdf:type rdf:Property.
   }
}
```

SPARQL query 9: Source Index Query (SourceIndexStrategy2)

5.5 Query analysis

This section discusses the mechanism used to analyze SPARQL queries, used to determine which sources are relevant for a particular query. Due to limitations of existing RDF frameworks for mobile devices, local (SPARQL) query analysis is not possible therefore the query analysis is offered as a remote service. To analyze SPARQL queries the SPARQL Engine for Java[24] is used, which is described in the first subsection. A second section explains how this API is used to help determine that relevant sources for a particular query.

5.5.1 SPARQL Engine for Java

The SPARQL Engine for Java is an API intended for RDF server implementations to be able to add SPARQL querying capabilities. It frees server implementations from details concerning query specifications and implementing these mechanisms themselves; it also accommodates users who are interested in further extending the SPARQL query language.

Within the context of this thesis, this API is used to construct a list of predicates used in a SPARQL query, and to compose a Source Index Query that can be used to query our Source Index Model to allow us to determine the relevant sources for a particular query.

To allow SPARQL query analysis, we utilize the API's SPARQLPARSER class to construct an Abstract Syntax Tree of a SPARQL query. To perform operations on the Abstract Syntax Tree,

[24] http://sparql.sourceforge.net/

the developer uses the Visitors Pattern[25]. The SPARQLParserVisitor is the Visitor interface and the constructed Abstract Syntax Tree is the object structure that can be visited within the Visitors pattern. By creating a concrete implementation of the SPARQLParserVisitor it becomes possible to perform operations on the constructed Abstract Syntax Tree.

A drawback of this API is that it is not intended for java ME and that it does not work within SCOUT framework itself. Since the actual parsing of a SPARQL queries falls out of the scope of this thesis, we used the sparQL Engine on a remote server and provide query analysis as a remote service. Later on the SPARQL parser functionality will be developed within the SCOUT framework itself.

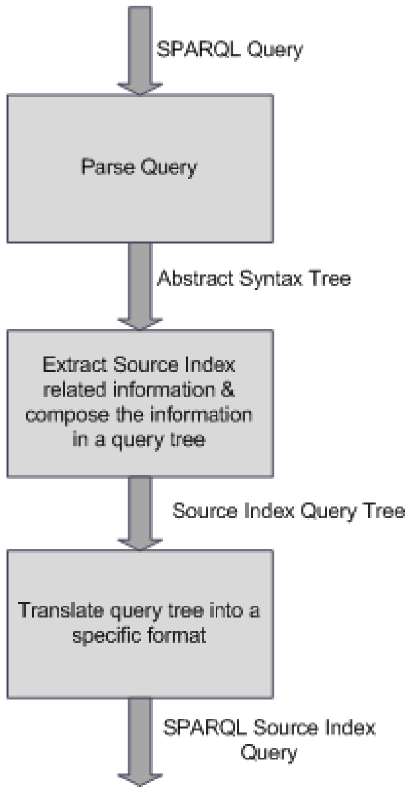
5.5.2 Query analysis to determine the relevant sources

A first aim of this thesis was to optimize the query process of the Environment Model. A Source Index Model is used, to determine what sources are relevant for a particular query, only these relevant sources are queried instead of all the encountered sources as was previously the case. This Source Index Model contains what kind of information (e.g., predicates) is used in the sources, in order to use this model to determine the relevant sources, a special query, called a Source Index Query, is used. This Source Index Query is constructed from the given query; the result of this query is all the relevant source for the given query. This section explains how this happens and how the SPARQL Engine for Java API is a part of this process. Section 7.4.3 provides the implementation details of this process.

Workflow 1 shows the process of constructing the Source Index Query from the given query (i.e., a SPARQL Query). The process of constructing the Source Index Query is a complex process, and consists of three main steps: 1/ constructing the Abstract Syntax Tree (AST) of the query using SPARQLParser; 2/ constructing from the AST a SourceIndexQueryTree (see section 5.5.3) and 3/ constructing from the SourceIndexQueryTree an actual query in the appropriate format, which depends on the format of the Source Index Model (e.g. SPARQL, SQL …).

The reason why we transform the AST into SourceIndexQueryTree (and not directly into an appropriate query string) is a matter of abstraction. Remember that the sparQL Engine is not intended for Java ME and thus cannot be used in the SCOUT framework itself. This means that at a later point in time, when a SPARQL parser exists for Java ME and has its own AST of a query, we only need to change the first two components and not all three of them.

[25] http://www.dofactory.com/Patterns/PatternVisitor.aspx


SPARQL Query
Parse Query
Abstract Syntax Tree
Extract Source Index related information & compose the information in a query tree
Source Index Query Tree
Translate query tree into a specific format
SPARQL Source Index Query

Workflow 1: Constructing Source Index Query

5.5.3 Source Index Query Tree

The previous section explained the process of forming a Source Index Query for a given query. One of the steps is to construct a Source Index Query Tree, this section explains the composition of WHERE clause this Source Index Query Tree. Firstly the design details are explained, next one example SPARQL queries illustrate the composition of the WHERE clause of a Source Index Query for Source Index Strategy 1 and Source Index Strategy 2.

The WHERE clause is represented as a where tree for which the Composite pattern[26] is used, with UnionNode and CompositeWhereNode as composites and WhereTripleNode as a leaf. The difference between CompositeWhereNode and UnionNode is that CompositeWhereNode represents a number of where constraints, while UnionNode represents a union of two or more groups of where constraints. We explain the process of extracting relevant WHERE clause information at hand of one example.

Consider SPARQL query 7, which selects the name of all person known by ‘Elien Paret’ and SourceIndexStrategy1, which contains information about the used predicates and their

[26] http://www.dofactory.com/Patterns/PatternComposite.aspx

corresponding actual domains. In this query two predicates are used: "`foaf:name`" and "`foaf:knows`", these predicate are each time used with the domain "`foaf:Person`". In our Source Index Query we will thus want to select all the sources that use either "`foaf:name`" with the domain "`foaf:Person`" and / or "`foaf:knows`" with the domain "`foaf:Person`". For each of the combination predicate-domain, we will use six whereTripleNodes, four to express that the predicate-domain combination must be present in the source and two to express that the domain and predicate are respectively of the type "`rdfs:Class`" and "`rdf:Property`".

Figure 8 visualizes the composition of the elements. The six whereTripleNodes are composed in a CompositeWhereNode. Since it is sufficient that a source uses one of the combinations, the CompositeWhereNodes are maintained in a UnionNode. Suppose that a source should contain both combinations, then all (twelve) WhereTripleNodes would be contained in one CompositeWhereNode.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE
{
   ?p1 foaf:name 'Elien Paret' .
   ?p1 a foaf:Person .
   ?p1 foaf:knows ?p2 .
   ?p2 foaf:name ?name .
   ?p2 a foaf:Person .
}
```

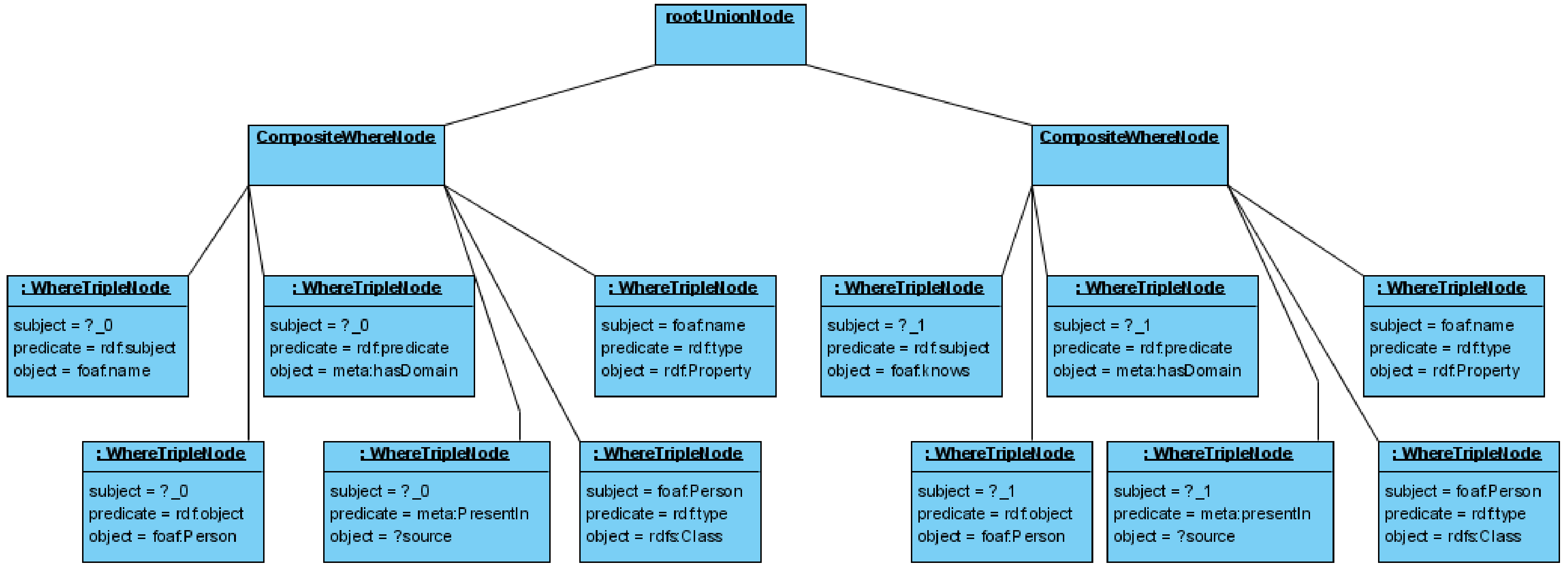


Figure 8: composition where tree (SourceIndexStrategy1)

Figure 9 shows the composition of the elements of the whereTree for SourceIndexStrategy2. Notice that this three is far less complex than the one of SourceIndexStrategy1, since the domain is no longer included. This time two WhereTripleNodes are used to select the sources using the predicate "`foaf:name`" and two for the predicate "`foaf:name`", one for saying that the predicate is a "`rdf:Property`" and one for saying that the predicate must be used in a source. Again the WhereTriplesNodes for one predicate are contained in a CompositeWhereNode and the CompositeWhereNodes are contained in a UnionNode.

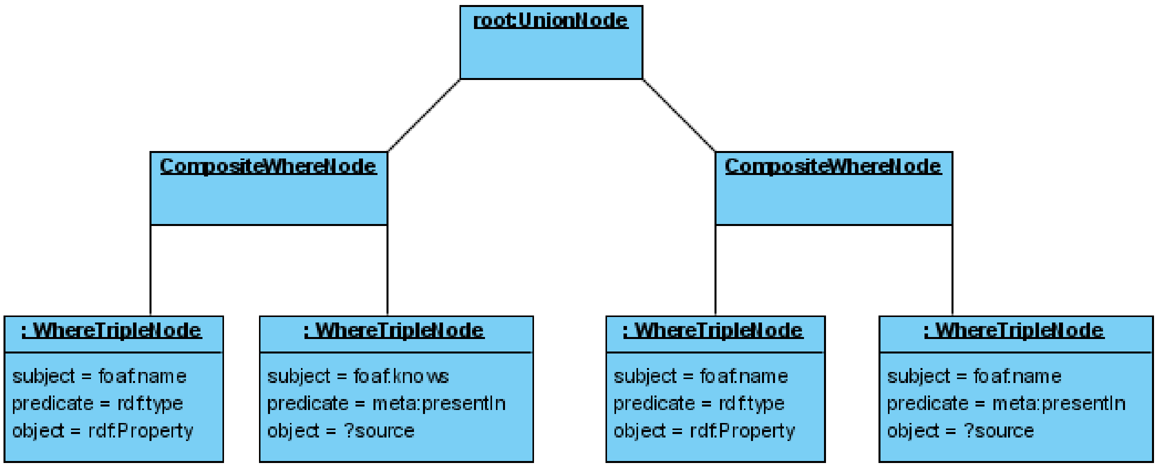


Figure 9: composition where tree (SourceIndexStrategy2)

5.5.4 Limitations

Currently the approach of composing a Source Index Query out of a SPARQL Query is not complete. It only takes the regular "`WHERE`" constraints into consideration; it does not consider the fact that "`FILTER`" clauses may also specify relevant predicates. To illustrate this, consider SPARQL query 10. This query uses a filter to specify a certain predicate (i.e., "`foaf:firstName`" and "`foaf:givenName`"). Currently, our query analysis process does not detect that "`foaf:firstName`" and "`foaf:givenName`" are used predicates. We do point out however that this is not a functional flaw: SPARQL query 10 can be rewritten as an equivalent SPARQL query shown in SPARQL query 11, which our approach is able to handle correctly. For a further elaboration on the use of "`FILTER`" clauses, please see Chapter 9.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE
{
   ?person ?namePred ?name .
   FILTER
   (
      sameTerm(?namePred, <http://xmlns.com/foaf/0.1/firstName>)
      ||sameTerm(?namePred, <http://xmlns.com/foaf/0.1/givenName>)
   )
}
```

SPARQL query 10: Filter query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE
{
   {
      ?person foaf:firstName ?name .
   }
   UNION
   {
      ?person foaf:givenname ?name .
   }
}
```

SPARQL query 11: Alternative filter query

Chapter 6 Cache

The aim of this thesis is to investigate how the Environment Model can be constructed and managed more efficiently. The first step was to maintain a Source Index Model that contains source index information telling us what kind of information is used in a specific source. A second step is too no longer store all of the encountered data sources locally, as was currently the case. Since it is totally unrealistic to assume that our mobile devices will have enough space to store all the sources.

This section explains the concept of the caching mechanism. The first section describes what is stored in the cache, from this exposition two variations of the caching mechanism are chosen. The first one caches entire sources, while the second one caches RDF triples using specific predicates. The second section explains the chosen eviction strategy (i.e., a strategy which determines what elements are evicted from the cache when it is full) in detail.

Before going to the details, we assumed that the encountered data does not change over time. This means that our caching system does not have a policy to assure the freshness of the cached data. It is considered Future Work to further develop the caching mechanism to include a freshness policy and to take into account the fact that the encountered data can change (see 9.2).

6.1 What to cache?

As discussed in the background part, different types of information can be stored in a cache, e.g., rows of a database table, query results, hypermedia-objects, etc. Considering that the information encountered within the SCOUT framework is RDF information, stored in a distributed manner, the cache should contain RDF or fragments of RDF data: RDF (sub) graphs, triples, part of triples, etc.

Our approach is to cache RDF triples. The granularity of the cached triples can vary: a cache entry may contain a single triple, all the triples of a particular source, all the triples using a certain predicate, all triples with a particular object or subject… As in the setting of SCOUT we continuously encounter sets of RDF triples in the form of online sources, the most straightforward strategy is to cache an entire encountered source. This means a one-to-one mapping of the relevant source and a cache entry. This possibility is designed (section 6.1.1), implemented (section 7.6.1) and evaluated (section 8.3).

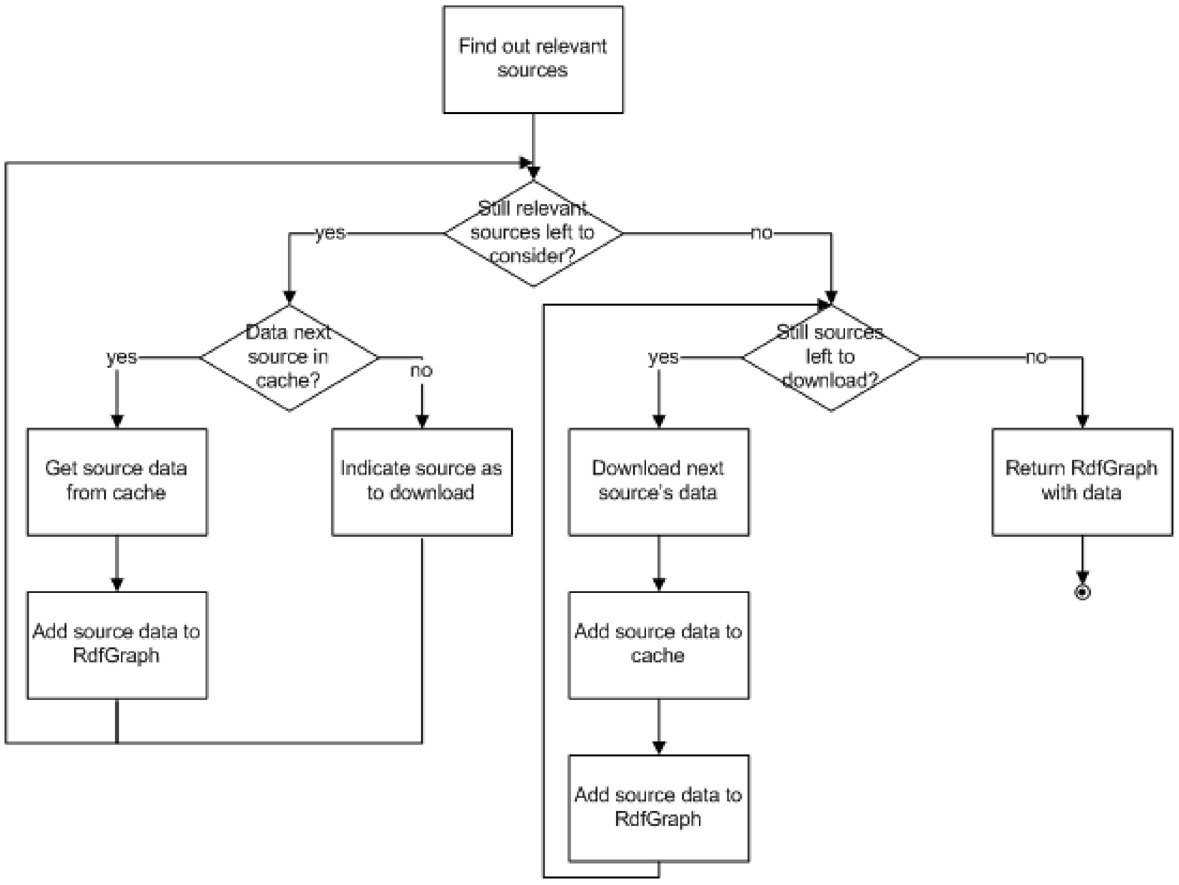
Since RDF information is by design predicate-oriented, i.e. it consists of triples of the form <subject> <predicate> <object>, a second strategy is to cache the RDF triples based on the predicate they use. Deciding to cache all triples using a specific predicate in one cache entrance in an RDF system seems as natural as caching all the information about a specific object in one cache entrance in an Object-Oriented system. This option is also fully designed (section 6.1.2), implemented (section 7.6.2) and evaluated (section 8.3).

6.1.1 Cache entire sources

The most straightforward strategy is to cache an entire encountered source. This provides a one-to-one mapping of the relevant source and a cache entry. The following workflow explains how this approach retrieves all the relevant information for a query posed to the Environment Model.

The first step is finding out the relevant sources for the particular query, this happens by using the source index strategy explained in section 4.1. For each relevant source, the algorithm checks whether the relevant source is cached. If so, it is immediately retrieved from the cache and added to the RDF Graph, which will contain all the RDF information of the relevant sources. This RDF Graph is used to resolve the query. If a source is not cached, it will be downloaded in a later step of the algorithm.

When all cached relevant sources are retrieved from the cache, the rest of the relevant sources (the uncached ones) are one by one downloaded, added to the cache and added to the RDF Graph. Benefit from downloading the not cached sources at a later point it time, is that it is more efficient in terms of memory usage. Suppose we download the relevant source at the point in the algorithm where we check whether or not the source is cached, then we would have to store them in memory until all of the cached relevant sources are retrieved from the cache. If we would immediately download and add them to the cache, we could evict a cached relevant source that has not yet been retrieved from the cache. So by downloading and adding the not cached sources at a later point, we avoid that we evict sources that are not retrieved from the cache yet and that we need to store them in memory until all the cached relevant sources are retrieved from the cache.


Find out relevant sources
Still relevant sources left to consider?
yes
no
Data next source in cache?
yes
no
Get source data from cache
Indicate source as to download
Add source data to RdfGraph
Still sources left to download?
yes
no
Download next source's data
Add source data to cache
Add source data to RdfGraph
Return RdfGraph with data

Workflow 2: Caching entire sources, retrieving the relevant information for a given query

In our setting new sources can and will be encountered at any time. This particular strategy does not add newly encountered sources to the cache since we do not expect them to be immediately accessed, queried.

6.1.2 Cache triples using a specific predicate

The previous section explained the straightforward cache strategy in which entire RDF sources are cached. A second cache strategy caches RDF triples using a specific predicate. Deciding to cache all triples using a specific predicate in one cache entrance in an RDF system seems as natural as caching all the information about a specific object in one cache entrance in an Object-Oriented system. This strategy has a smaller granularity than the first straightforward strategy. As a consequence, this cache requires less storage, uses the cache storage space more efficiently and more relevant information is stored in the cached. A second benefit is that a given query is executed on an RDF Graph which only contains information about the predicates used in the query and about nothing else; contrary to the RDF Graph of the previous cache strategy, where the RDF Graph contained also other RDF information. Consequently, the resulting combined RDF Graph over which the particular query needs to be executed is smaller, yielding faster query time and less memory is required. A drawback of this approach is the amount of overhead that is introduced to extract triples of a particular predicate from an encountered source. Section 8.3 evaluates the impact of this drawback.

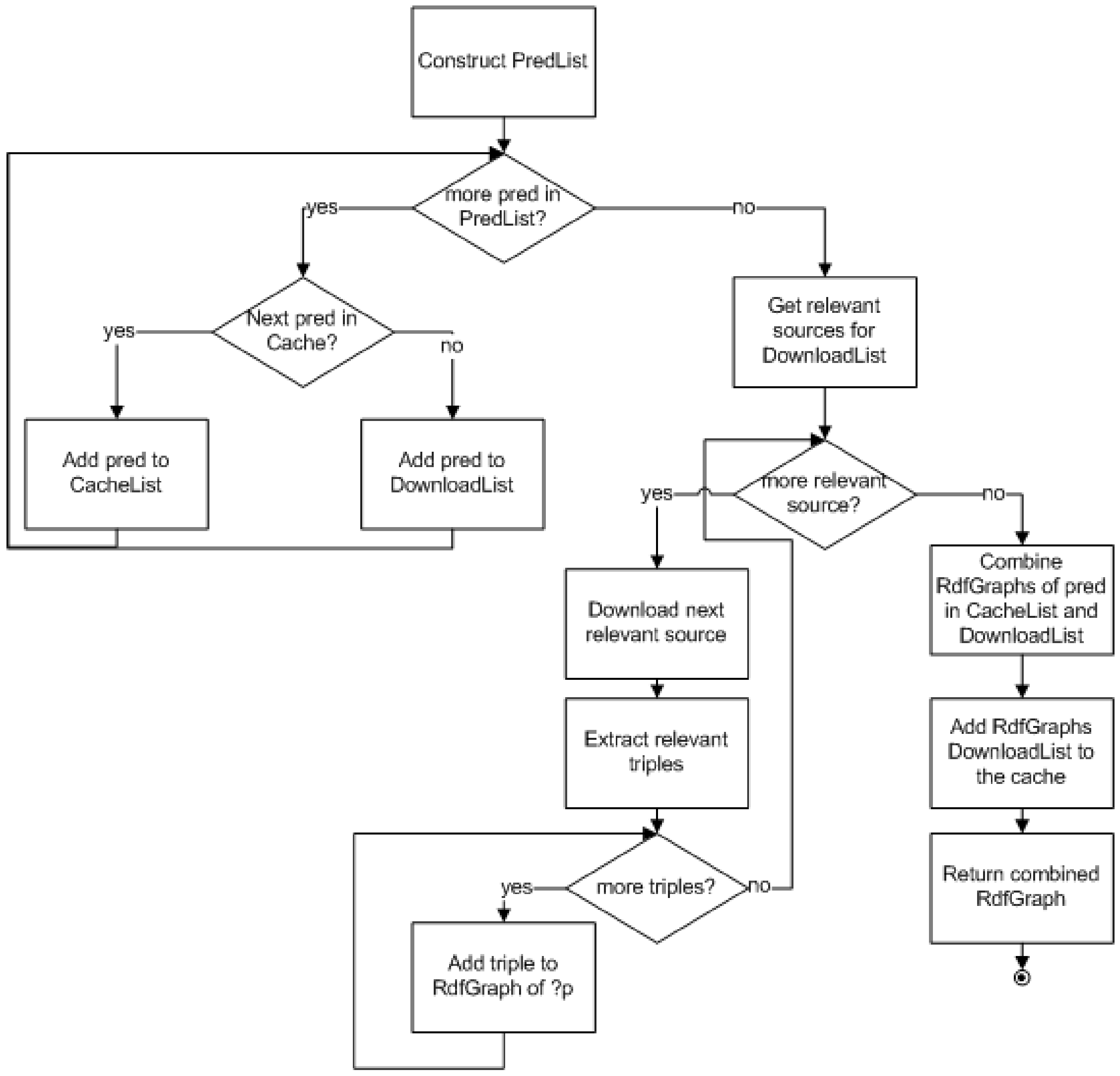
This section firstly describes the general principle of what should happen when a query is posed to the Environment Model. Secondly the complicating factors are discussed followed by the final variant. The last paragraph describes what happens when a new source is encountered.

Workflow 3 shows the actions that occur when a query is posed to the Environment Model this workflow shows the general idea and not the final variant.

The first step of the general process is collecting the predicates that are used in the query (i.e., constructing a PredList in the workflow). Once these predicates are known, they are divided into two groups: a first group of cached predicates (i.e., CacheList in the workflow) and a second group of not cached predicates (i.e., DownloadList in the workflow), the RDF information for this second group will need to be downloaded and constructed.

The second step is to collect the RDF information, i.e., RDF triples, for all the predicates that are not present in the cache. In order to do so, it is determined (by using a Source Index Strategy) which sources contain triples using one of those predicates. Secondly, each of the relevant encountered sources is downloaded and the triples using one of those predicates are extracted from the source and put into the RDF Graph of the corresponding predicate. After all the relevant sources are downloaded and the relevant triples are extracted from them, the result is one RDF Graph for every predicate (relevant for the query) that contains all the triples that use the specific predicate.

The next step is to construct the overall RDF Graph, by combining the cached predicates with the newly downloaded predicates. Before returning the RDF Graph with all the relevant data, and in order to allow usage of the most recent version of the cache for other following queries, the cache is updated first. Updating the cache means simply adding the RDF Graph of non-cached predicates to the cache.



Workflow 3: Caching triples, getting the information for a particular query (general idea)

As previously mentioned, there are some (RDF specific) issues that complicate the general principle. The problem lies in the peculiarities of blank nodes in RDF. This section discusses this complicating factor in detail and uses some examples to explain the problem.

A blank node in the context of RDF is a node that has no global identifier. In practice, RDF APIs mostly generate a random identifier to make it possible to address the blank node. However, this identifier is only unique within the RDF Graph itself, and not across several RDF Graphs or even several instantiations of the same RDF graph; in other words, executing the same query twice on the same RDF file will result in different blank node identifiers for the same nodes. RDF data 12 is an RDF example in N-Triple format that contains three statements about a blank node. Blank nodes in the N-Triple format always start with "`_:`" followed by their identifier.

Our algorithm extracts nodes (subjects and objects) from their original source file by executing an extraction query, and puts them into a new RDF Graph (i.e., the predicate RDF Graph). However, the blank node identifiers of these nodes must still correspond to the identifiers of the same nodes already present in other predicate RDF graphs. As mentioned above this cannot be guaranteed, as the latter nodes were extracted via another execution of an extraction query. Moreover, as blank node identifiers are mostly generated randomly, it is (at least theoretically) possible that two different nodes, from the same or different sources, are assigned the same blank node identifier. Two examples are given below which illustrate these issues.

For instance, consider the case where two predicate RDF graphs are constructed, one for "`foaf:name`" and one for "`foaf:givenname`". When forming an RDF Graph for the predicate "`foaf:name`" , the blank node associated with that predicate in source B will get a unique identifier from the query resolver (e.g., bn_1). Afterwards, the RDF Graph for the predicate "`foaf:givenname`" is constructed, and the necessary triples are extracted via a different extraction query execution; consequently, a new identifier for the blank node in source B is generated (e.g., bn_2). When these two RDF Graphs are merged at a later stage in order to solve a given query, the same resource will appear as two different blank nodes: one with identifier bn_1 and one with identifier bn_2.

```
…
_:n1  <http://www.w3.org/2000/01/rdf-schema#seeAlso>
 <http://wise.vub.ac.be/members/peterp/foaf.rdf> .
_:n1 <http://xmlns.com/foaf/0.1/name> "Peter Plessers" .
_:n1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
…
```

RDF data 12: source A

```
…
_:n1 <http://xmlns.com/foaf/0.1/name> "Pieter Callewaert" .
_:n1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
_:n1 <http://xmlns.com/foaf/0.1/givenname> "Pieter" .
_:n1 <http://xmlns.com/foaf/0.1/nick> "Pierre" .
…
```

RDF data 13: source B

As a second example, suppose that in the processes of getting data for a particular query, the information of the predicate "`foaf:name`" (present in source A and source B) needs to be stored in the cache. The process will construct an RDF Graph for the predicate "`foaf:name`", where the triples "`_:n1 <http://xmlns.com/foaf/0.1/name> "Peter Plessers"`" "`_:n1 <http://xmlns.com/foaf/0.1/name> "Pieter Callewaert"`" are extracted from source A and B, respectively (see RDF data 12 and RDF data 13). However, the two different subjects were incidentally assigned the same identifier by our SPARQL query resolver (i.e., `_:n1`); therefore, as far as the RDF Graph is concerned, the two triples are about the same node since the two nodes have the same identifier.

Comparing the results of different query executions on the same source, has no use since it is not guaranteed that blank node identifiers are the same across different instantiations of the source (and thus they are very different across different executions of the same query). Consequently you cannot know whether one blank node matches another (previously extracted) blank node from a predicate RDF Graph, since there are false positives (i.e., two different nodes with the same identifier) and false negatives (i.e., two equal nodes with a different identifier). An idea would be to determine the equality of two nodes by using the predicate and value, however this will not work since two different blank nodes can have the use the same predicate and object. However in one query, the same blank node is always identified by the same identifier. The only option is to include the cached predicates that use blank nodes from the same source, in the Source Index Query to have the same blank node identifiers. To avoid that different nodes get the same identifier by coincidence, the unique identifiers are given by the Source Index Strategy.

Consequently, in order to find out which (already cached) predicates need to be re-included in the extraction query, it must be known for each cached triple from which source it originated and whether it contains a blank node. To achieve the former, extra information thus needs to be stored. One immediate solution is to work with reified statements (i.e., statements that can be referred to); that way, triples can be added to the predicate RdfGraph that keep the origin source for each original (reified) triple statement. RDF data 14 shows what data must be kept to state that the following triple "`<http://wise.vub.ac.be/members/sven/foaf.rdf#me> <http://xmlns.com/foaf/0.1/firstname> "Sven"`" occurs in the source "`http://wilma.vub.ac.be/~eparet/elien_foaf_N.rdf`". One reified statement thus results in four new statements, with one additional triple keeping the origin source for the reified statement. This means that the storage space for keeping this one extra piece of information is multiplied by five, which is clearly unacceptable.

```
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffc
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffc
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
      <http://wise.vub.ac.be/members/sven/foaf.rdf#me> .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffc
```

```
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
      <http://xmlns.com/foaf/0.1/firstname> .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffc
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
      “Sven”.
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffc
   <meta:presentIn>
      <http://wilma.vub.ac.be/~eparet/elien_foaf_N.rdf> .
```

RDF data 14: A ReifiedStatement, the source of a triple

Another possibility is to no longer store the information (the extracted triples using a particular predicate and the source from which they were extracted) as an RDF Graph, but as a special predicate list. For each triple the subject, object, types of subject and object (i.e., URI node, blank node or literal), and the origin source for a specific triple is stored. So five characteristics are stored for each triple, these five characteristics of a triple are called TripleElements. The object and subject types are needed to know the type (blank node, literal or URI) of a subject or object, which is necessary in the recomposition of the RDF Graph as each node type needs to be created in a specific way and to know which triples use a blank node. Note that the predicate of the triple is not stored in the TripleElement, since all the triples in a predicate list obviously use the same predicate.

An obvious drawback of this solution compared to the reified-statement solution (as is illustrated in RDF data 14) is that the RDF Graph needs to be decomposed into the list structure whenever it is stored in the cache, and again recomposed when the RDF information is necessary to execute a given query. However, regarding space requirements this solution is much more acceptable than the reified-statement solution.

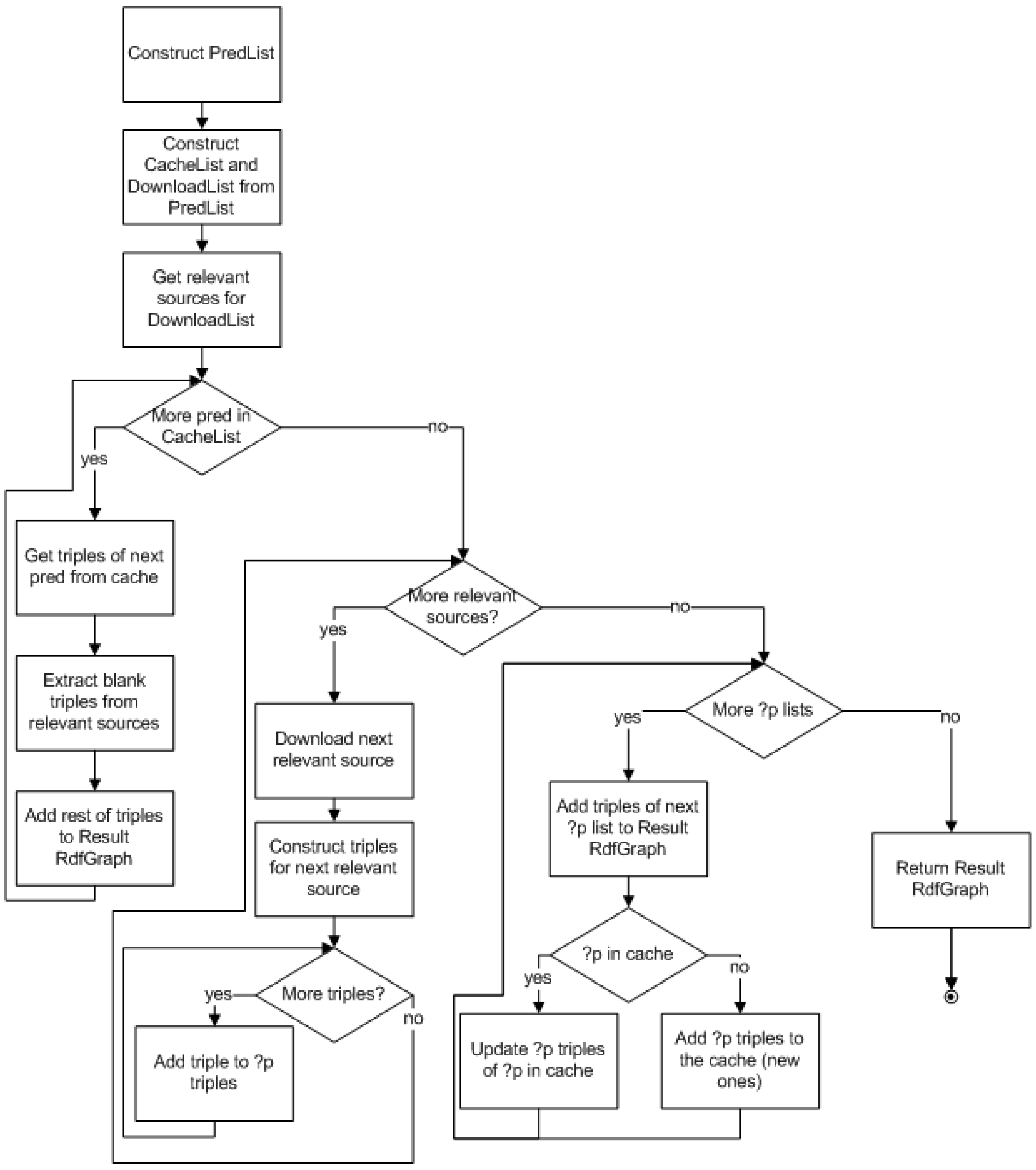
The conclusion from this section is that blank nodes significantly complicate the general principle of CacheStrategyPredicate, and as a result, a list must be stored instead of an RDF Graph for each predicate. Furthermore, every time triples are extracted from a source for which the cache already contains triples, the blank node identifiers from the source and the cache (if any) need to be synchronized to avoid node identification issues.

Keeping the conclusions in mind, Workflow 4 visualizes the final variant; the beginning of the workflow is the same as for the general idea: construct the PredList and split it up in a CacheList, containing the predicates which are cached, and a DownloadList, containing the predicates that are not stored in the cache and thus need to be downloaded.

The next step is retrieving the relevant sources for the predicates in DownloadList. Before actually downloading the relevant sources and extracting the relevant RDF triples, the actual variant first needs to find out which cached predicates need to be re-included in the extraction query, with the goal of synchronizing the blank node identifiers from the cache with the extracted blank nodes. For this purpose, it retrieves the cache elements (i.e., list of TripleElements) for all the cached predicates, and checks which ones contain blank nodes and originate from a source on which the extraction query will be executed. Subsequently, the corresponding extraction queries are altered based on the extra predicates that need to be included. In a later step, the blank node identifiers for these TripleElements will be updated using the extraction query results (see later). Furthermore, the TripleElements (actually, the “real” RDF triples corresponding to these elements) that do not contain blank nodes and are not from relevant sources, are added to the final RDF graph: i.e., the graph on which the query will be executed. Note that these elements must be obtained *before* updating the identifiers of blank nodes in the cache, since the update method of the cache can remove aged and least recently used elements and thus could result in evicting these elements from the cache.

After adding the cached information to the final RDF graph, the relevant sources are downloaded. In the general idea, the triples that use a predicate from DownloadList can be extracted from the (source) RDF Graph and directly stored per predicate in an RDF Graph. However, in our case the result triples from the extraction query also include triples for which the blank node identifier needs to be updated in the cache.

The following step is combining the triples obtained from the cache and the results of the extraction queries. In the actual variant this step is more complex, since the set of RDF triples (belonging to a predicate) stored in the cache is no longer stored as an RDF Graph, but as a list of TripleElements. This means that an RDF Graph must first be constructed for each list. Later on, these RDF Graphs are combined into a single RDF Graph, which will be returned. After this step, all the relevant information of the particular query is contained in the return RDF Graph. During the construction of the final RDF graph, the relevant TripleElements are updated with their new blank node identifiers, as obtained from the extraction query results, and triples of new predicates are added to the cache. Finally the RDF Graph is returned.


Construct PredList
Construct CacheList and DownloadList from PredList
Get relevant sources for DownloadList
More pred in CacheList
yes
no
Get triples of next pred from cache
Extract blank triples from relevant sources
Add rest of triples to Result RdfGraph
More relevant sources?
yes
no
Download next relevant source
Construct triples for next relevant source
More triples?
yes
no
Add triple to ?p triples
More ?p lists
yes
no
Add triples of next ?p list to Result RdfGraph
?p in cache
yes
no
Update ?p triples of ?p in cache
Add ?p triples to the cache (new ones)
Return Result RdfGraph

Workflow 4: Caching triples, getting the information for a particular query (actual variant)

In order to construct a list of predicates that are used in a SPARQL query, the query is analyzed by using the SPARQL Engine for Java[27] (see section 5.5.1). Firstly we use the SPARQLParser class to build an Abstract Syntax Tree (AST) from the SPARQL query. Secondly we go over the

[27] http://sparql.sourceforge.net/

AST and extract the used predicates and collect them in a list. Since this API is not available for a mobile setting, currently this is offered as a remote service.

After looking at the complex details of what happens when the cache strategy must construct the relevant data for a query posed to the Environment Model, this paragraph describes what happens when a new source is encountered. Contrary to the previous strategy in which entire RDF sources are cached and nothing happens when a new source is encountered, this strategy performs actions to assure the correctness of the cache, to assure each list of triples per predicate in the cache is complete; i.e., it contains all the triples using that predicate from all encountered sources and thus also for the newly encountered one. This means that when a new source is encountered, all the triples of the newly encountered source that use a cached predicate are added to the corresponding cache elements, i.e., to the list of the corresponding predicate. If we don't do this, the next time the predicate information is retrieved from the cache, it will not contain the relevant triples for that predicate from our newly encountered source. In this process the unique identifiers of the blank nodes are taken into consideration, in other words, it is ensured that a blank node from the source gets an identifier that is unique across all the cached information.

6.2 Eviction strategy

The previous section discussed what type of information is stored in the cache; this section covers the eviction strategy that is chosen to decide which cache element(s) can stay in the cache, and which ones are removed in case the cache is full. A couple of common removal strategies were previously discussed in the background chapter (Chapter 2): Least Recently Used (LRU), First In First Out (FIFO), and Least Frequently Used (LFU)… Depending on the expected access pattern of the information, one algorithm is more appropriate than the other.

In the SCOUT framework, it is expected that access to the different information sources provided by the different encountered entities will not be sequential: sources will rather be accessed depending the type of information they hold (e.g., persons, locations) or the timeframe wherein they were encountered. As a consequence, we can rule out the First In First Out eviction strategy.

Least Frequently Used (LFU) could be a suitable approach for our caching mechanism. However one needs to consider that the number of accesses to all the sources (also those that are not in the cache) must be tracked, which causes some overhead. A more important reason for not choosing LFU as an eviction strategy is that we expect that the client will access the same kind of information in a short timeframe and this algorithm has the characteristic to keep sources in cache that were very frequently over time, not necessarily in the present.

Least Recently Used (LRU) is the chosen eviction algorithm. It is a fast algorithm with minimal overhead (important in our real-time system) that uses the concept of recentness to decide what information must stay in the cache. To illustrate why recentness is important in our setting, consider the following typical scenario for mobile users, which illustrates our expectation that the client will access the same sort of data in a specific timeframe before looking for other type of data. This expectation makes LRU the eviction strategy of choice for the SCOUT framework.

Consider a client called Pieter who is in Brussels and wants to go to a restaurant. Pieter decides to use his mobile phone to help him find a restaurant that is nearby his current position. His mobile application “Restaurant Finder” shows a list of restaurants that are nearby. Pieter decides to narrow his search by saying that the restaurant must serve pasta; a shorter list of restaurants appears on his screen. He sees the restaurant “Pastage” and decides to view the details of the restaurant. Looking at the details Pieter is not convinced that “Pastage” is what he wants. Pieter navigates back to the overview. Now he sees the restaurant “La Dolce Vita”, after looking at the details, he decides that “La Dolce Vita” is the place to be. He asks his route planner to take him to “La Dolce Vita” and he enjoys a very good pasta dish. This scenario illustrates that recently used information (i.e., nearby restaurant) is queried repeatedly, while other information at that point is irrelevant.

When using the LRU algorithm one must consider what action will determine the recentness of the elements. In our setting asking for an element to cache and adding an element to the cache will determine the recentness of the element. So, the least recently used element is the element that was least recently asked or added to the cache.

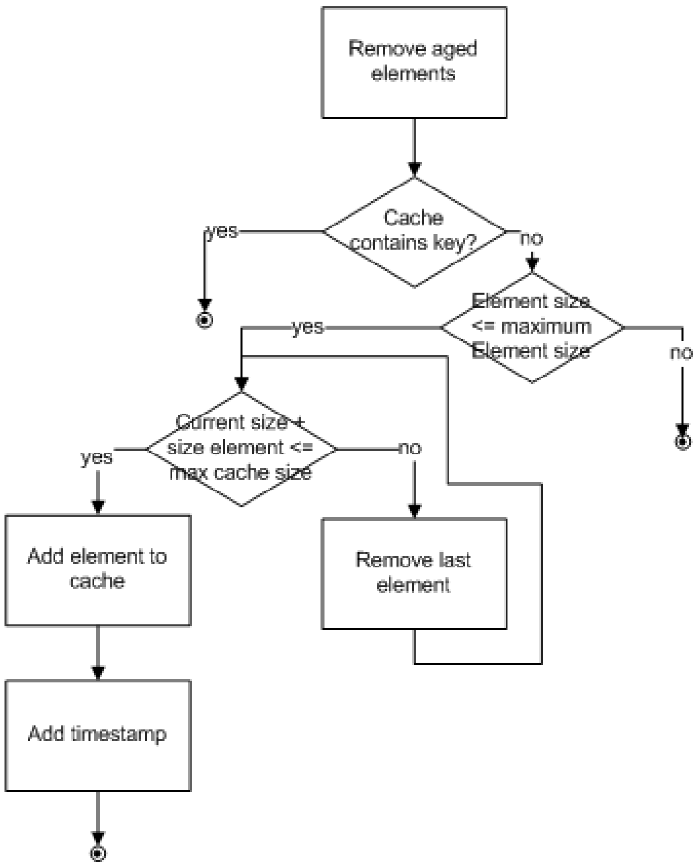
The LRU was not applied in its original form, it was optimized to remove the information that has been in the cache but not used for a very long time. By eliminating these elements while the cache is not actively in use, we reduce the amount of occupied space in favor of relevant information when the cache is active again. This happens by maintaining for each cache element a time stamp indicating when the element was last accessed.

The following subsections explain what happens when a cache element is requested, when it is added to the cache and when it is updated. As previously explained, our cache contains either the entire encountered source or all the triples using a specific predicate. In the first case, the cache uses the source URI as a key; in the second the predicate (i.e., a RDF resource defined by an IRI) is used as a key.

6.2.1 Adding an element to the cache

The workflow below explains the process of adding an element to the cache. Before adding an element to the cache three constraints are checked. First of all does the cache contain the key already? This is to avoid adding the same element multiple times. Secondly is the size of the element smaller than or equal to the allowed element size? This second constraint has as purpose to avoid that large elements take up all of the cache size. Thirdly is there enough space in the cache? If the cache is full our eviction strategy evicts the Least Recently Used element. At the end of the workflow, the cache element gets a timestamp to know when it was last used.

Before checking the constraints, the optimization, removing cached elements that are too long unused, of the LRU algorithm is performed. These elements are evicted from the cache in order to free up the cache space.

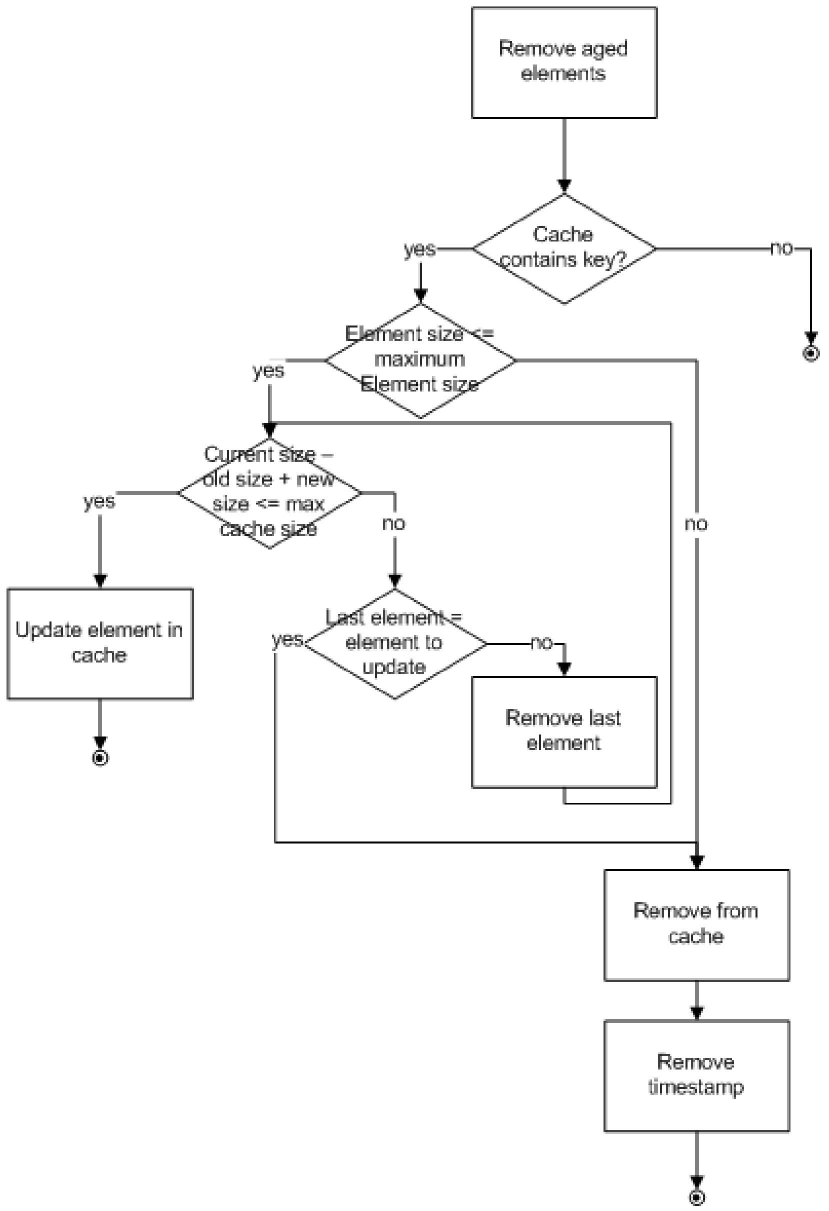


Workflow 5: Adding an element to the cache (LRU)

6.2.2 Updating an element from the cache

The following workflow shows the process of updating a cache element. This process checks the same three constraints as in the process of adding an element to the cache. Does the cache contain the element already? Is the element size smaller than or equal to the maximum element size? Will the maximum cache size be exceeded by updating the cache element? The optimization, evicting the aged elements of the cache, of LRU again happens before checking the three constraints.

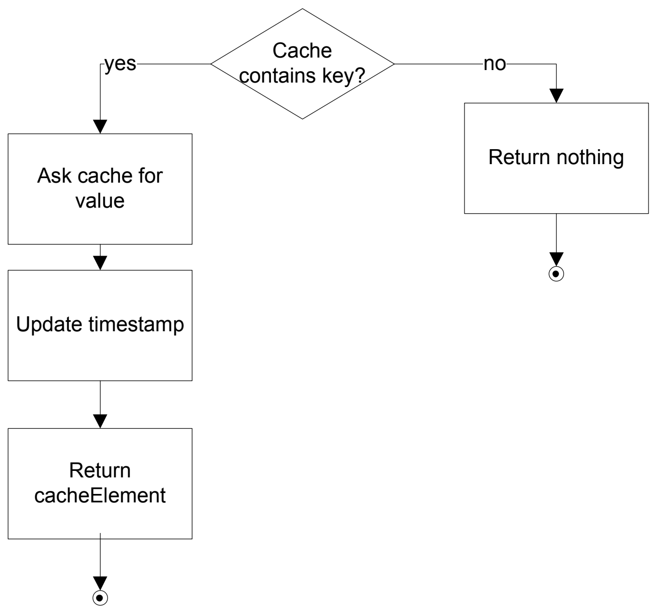
It is possible that during the eviction of the aged element or least recently used elements, the element to update is removed from the cache. In that case, the element will not be (re)added to the cache, since in our setting the get and add operation are the operations which determine which element is least recently used and thus which element should be evicted from the cache.



Workflow 6: Updating an element from the cache (LRU)

6.2.3 Retrieving an element from the cache

Workflow 7 shows the process for retrieving a specific element from the cache. During this process the aged elements are not removed from the cache, since the cache content does not change and it is thus not necessary to free up space in the cache. The timestamp of the cache element however gets updated since this is necessary to determine what elements are aged.



Workflow 7: Getting an element from the cache (LRU)

Chapter 7 Implementation

Previously the Environment Model has been stored as a fully materialized view; in other words, all of the data (i.e., encountered data sources) was kept locally. This thesis investigated how the Environment Model can be constructed and managed more efficiently, therefore two mechanisms are developed: 1) a mechanism responsible for maintaining the source index information, called Source Index Strategy, and 2) a caching mechanism to optimize the query solving and to reduce the amount of information that needs to be downloaded when queries are posed to the Environment Model.

This chapter explains the concrete implementation of those mechanisms in detail, while, Chapter 5 and Chapter 6 discussed the design. First of all the overall implementation is described (section 7.1). It explains which components are necessary to achieve our purpose, their specific function and the interaction between them. The latter sections all discuss a particular component in detail. Section 7.2 and 7.3 discuss management of the Environment Model, using caches and source index strategies. Section 7.4 discusses the different source index strategies that were used to keep track of online sources, and which information to store about them. Section 7.5 discusses caches and their working, applied to our specific setting. Section 7.6 discusses the caching strategies we applied in this dissertation. The final section (7.7) gives an overview of the design patterns that were used during the implementation.

7.1 Implementation Overview

The Chapter 4 explained how the two developed mechanisms, Source Index Strategy and Cache Strategy, fit into the SCOUT framework. This first section provides a similar overview but this time by showing the implementation details.

7.1.1 Accessing the Environment Model

Figure 10 shows a similar overview as Figure 7 but this time the implemented components are mentioned instead of the conceptual components. Figure 11 shows the concrete class diagram of the components, while Figure 12 shows the sequence diagram illustrating what happens when the Environment Model is accessed.

Figure 7 showed that the mobile application accesses its context and environment via the Environment Model, which is managed by the Environment Manager. The Environment Manager delegates the task of actually maintaining the encountered sources (i.e., the Source Index Model and Cache) to the Environment Access, while the Environment Manager forms and queries the actual Environment Model by accessing the Relation Manager, responsible for maintain the Relation Model, the Entity Manager, responsible for maintaining the Entity Model and the Environment Access.

The Environment Access uses a cache strategy to retrieve all the relevant information for a specific query, either stored locally or not. The cache strategy uses the cache component to actually store the information. This component determines which of the information, called cache elements, it stores according to a specific eviction strategy (in our case Least Recenlty Used). The cache strategy uses also the source index strategy, responsible for maintain the Source Index Model, to determine the relevant sources, so that he knows what information he must retrieve from the cache.

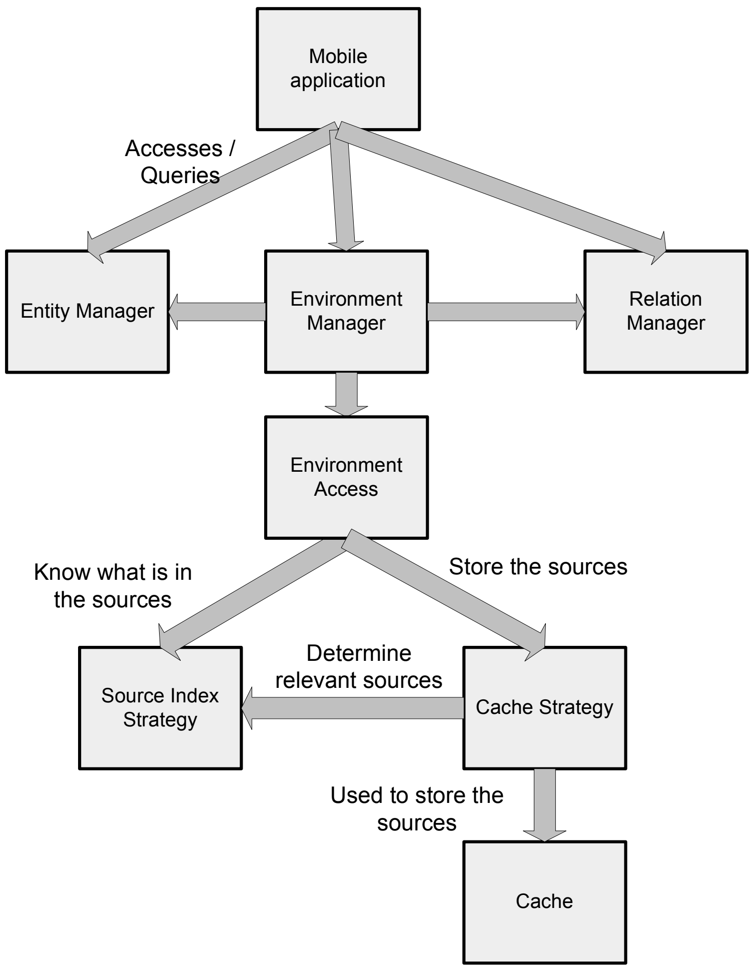
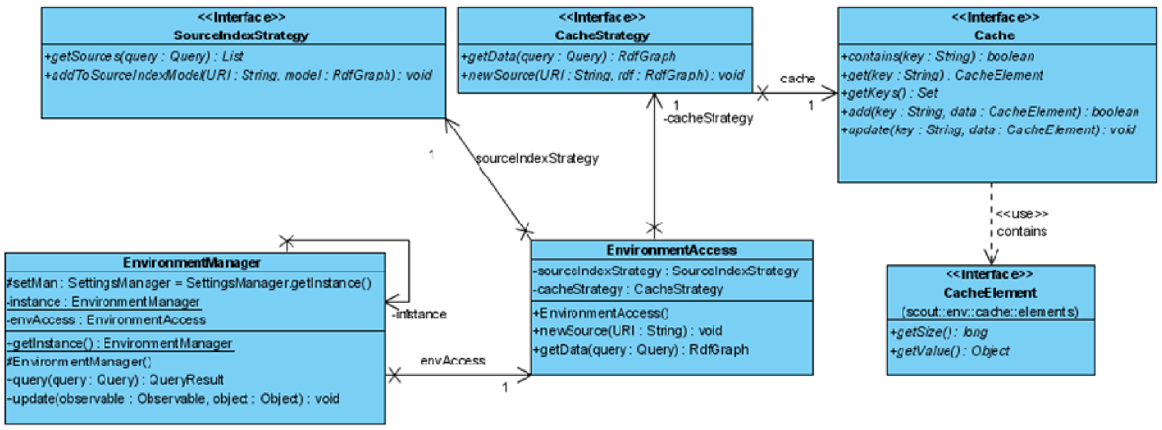


Figure 10: Accessing the Environment Model



<<Interface>> SourceIndexStrategy
+getSources(query : Query) : List
+addToSourceIndexModel(URI : String, model : RdfGraph) : void
<<Interface>> CacheStrategy
+getData(query : Query) : RdfGraph
+newSource(URI : String, rdf : RdfGraph) : void
<<Interface>> Cache
+contains(key : String) : boolean
+get(key : String) : CacheElement
+getKeys() : Set
+add(key : String, data : CacheElement) : boolean
+update(key : String, data : CacheElement) : void
cache
-cacheStrategy
-sourceIndexStrategy
<<use>> contains
EnvironmentManager
#setMan : SettingsManager = SettingsManager.getInstance()
-instance : EnvironmentManager
-envAccess : EnvironmentAccess
-getInstance() : EnvironmentManager
#EnvironmentManager()
-query(query : Query) : QueryResult
-update(observable : Observable, object : Object) : void
-instance
envAccess
EnvironmentAccess
-sourceIndexStrategy : SourceIndexStrategy
-cacheStrategy : CacheStrategy
+EnvironmentAccess()
+newSource(URI : String) : void
+getData(query : Query) : RdfGraph
<<Interface>> CacheElement
(scout::env::cache::elements)
+getSize() : long
+getValue() : Object

Figure 11: Class diagram, accessing Environment Model

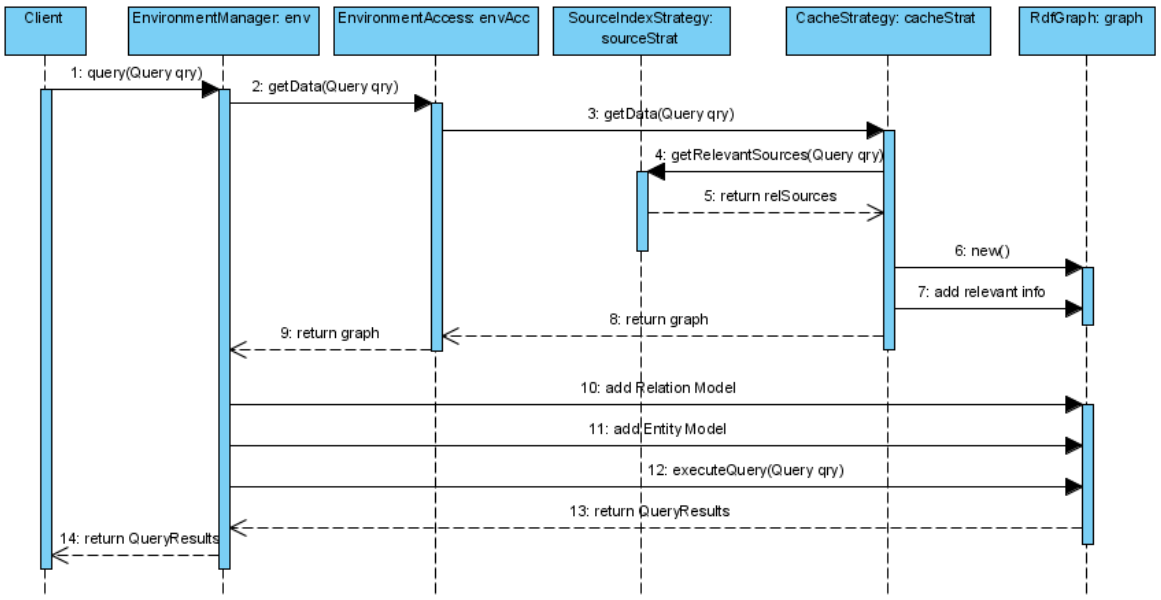


Figure 12: Sequence diagram accessing Environment Model

The last paragraph of this section gives some additional motivation for dividing certain functionality across several components. Firstly, by giving the Environment Access the responsibility to maintain the encountered sources, there is a looser coupling between the Environment Manager and the cache and source index strategy that is being used. The Environment Access initializes the appropriate cache and source index strategy and keeps them up to date. It allows the Environment Manager to only deal with the merging and querying of the different information (i.e., Relation-, Entity Model and encountered data sources). Secondly the source index strategy is captured in a separate component to provide the flexibility to easily change the used strategy and to be able to reuse the same logic to determine the relevant sources across several cache strategies. Thirdly the determination of what type of elements are stored in the cache and the eviction strategy are separated into the CacheStrategy and cache components. This to provide a looser coupling between the two components, it allows us to easily change the eviction strategy without the need to adjust the cache strategy which determines what kind of information is stored in the cache and to reuse a specific eviction strategy in while not having to deal with the details of the different type of elements that are stored in the cache.

7.1.2 Encountering new sources

From section 4.2 we know that the context and environment of our user changes frequently, at all times newly data sources are encountered and therefore the Environment Model has to be updated whenever a new data sources is encountered. Updating the Environment Model means updating the cache, the Source Index Model and the Relation Model. This section explains in implementation details what happens when a new source is encountered.

The SCOUT component that monitors and stores when a (new) source is nearby (i.e., encountered) or no longer nearby is the Relation Manager. This component can notify other components like the Environment Manager whenever the environment of the entity has changed. The Observer pattern is used to allow the notification of various components and allows determining at runtime which components need to be notified see 7.7 for more details about the implementation of the pattern.

The Environment Manager thus receives a notification of the Relation Manager whenever something has changed in the environment. The Environment Manager only notifies the Environment Access when a new source is nearby, it does not notify Environment Access of any other change (e.g., an entity is no longer nearby).

The Environment Access firstly downloads the RDF information at the URI location given by the newly encountered source, secondly it constructs an RDF Graph containing the downloaded information, thirdly it updates the Source Index Model, via the Source Index Strategy, and finally it offers the newly encountered source to the Cache Strategy. All these actions are illustrated as a sequence diagram in Figure 13.

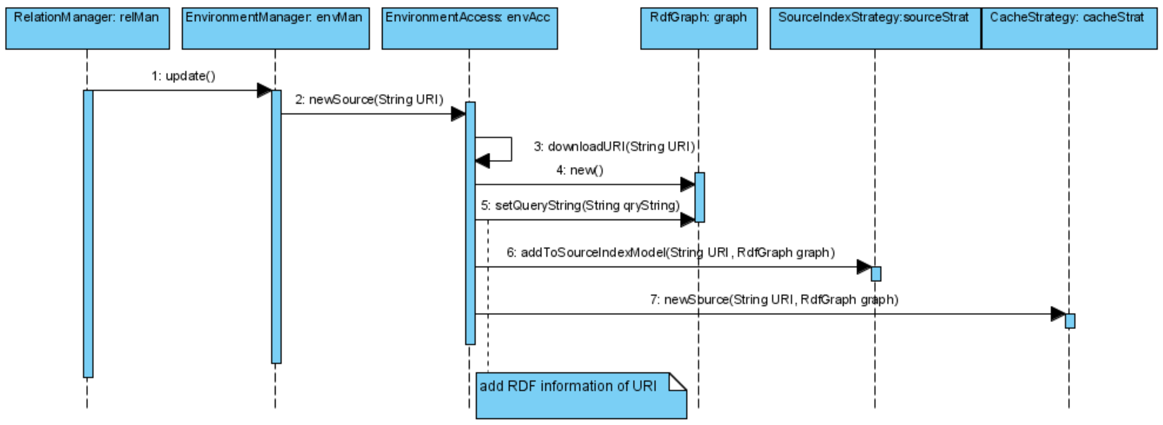

RelationManager: relMan
EnvironmentManager: envMan
EnvironmentAccess: envAcc
RdfGraph: graph
SourceIndexStrategy:sourceStrat
CacheStrategy: cacheStrat
1: update()
2: newSource(String URI)
3: downloadURI(String URI)
4: new()
5: setQueryString(String qryString)
6: addToSourceIndexModel(String URI, RdfGraph graph)
7: newSource(String URI, RdfGraph graph)
add RDF information of URI

Figure 13: Sequence diagram, encountering a new source

7.2 Environment Manager

Mobile applications developers will be able to developed mobile applications that are aware of their environment en the objects in it by using the SCOUT framework, which uses a model to capture the entire context and environment of the mobile user, called the Environment Model. This model is composed of the Relation -, Entity Model and the encountered data sources. The Environment Manager is responsible for providing access to this Environment Model by resolving queries over this model. The Environment Manager itself is not responsible for actually maintaining the Environment Model; therefore it uses the Relation Manager, Entity Manager and Environment Access. This section describes the implementation of this specific component (see Figure 11 for a detailed class diagram).

The state of the Environment Manager consists of three variables:

- SetMan (SettingsManager): instance which contains all the specific settings.
- Instance (Environment Manager): the unique instance of the Environment Manager, since only one Environment Model exists, the Single Pattern[28] is used to ensure that only one instance of the Environment Manager exists. For more information on the used design pattern see section 7.7 .
- EnvAccess (EnvironmentAccess): instance of the Environment Access component which is responsible for providing access and maintaining the encountered data sources.

The Environment Manager has three methods:

- getInstance: returns the unique instance of the Environment Manager

[28] http://www.dofactory.com/Patterns/PatternSingleton.aspx

- query: executes the specific query on the Environment Model and returns the query results. Therefore the Environment Manager asks the Relation Manager, the Entity Manager and the Environment Access for their information, combines their information, executes the query on it and returns the query results.
- Update: via this method, the Environment Manager knows that something its environment has changed, a new entity is encountered, a entity is no longer nearby … When the change in the environment is "encountered a new entity", he will notify the Environment Access, which keeps the model about the encountered data sources up to date.

7.3 Environment Access

The Environment Model, which captures the entire (previous and current) environment of the mobile user, is accessed via the Environment Manager. This component uses three other components, i.e., Entity Manager, Relation Manager and Environment Access, to maintain each a part of the environment. The Entity Manager contains all information, meta data, of the mobile user himself, the Relation Manager maintains the current and past relations with other entities while the Environment Access contains all the data comprised from the other encountered entities. This section explains the implementation details of the latter (see Figure 11 for a detailed class diagram.

The Environment Access has two fields:

- sourceIndexStrategy (SourceIndexStrategy): a strategy that is used to provide an accurate view on what information (e.g., what predicates) is used in the encountered data sources. This strategy can determine the relevant sources for a particular query by using this accurate view.
- CacheStrategy (CacheStrategy): a strategy responsible for deciding what information of the encountered data sources is kept locally and for providing access to the encountered sources. The Environment Access thus delegates the task of providing access to the encountered sources to the Cache Strategy component.

The Environment has two methods:

- NewSource: dictates the logic that is executed when a new data source is encountered. This means providing the URI of the encountered data source and the actual source content to the SourceIndexStrategy and CacheStrategy, so that they can up data their view on the encountered sources.
- GetData: is responsible for constructing a combined RDF Graph of all encountered relevant data sources. This happens by delegating asking cache strategy for the relevant information.

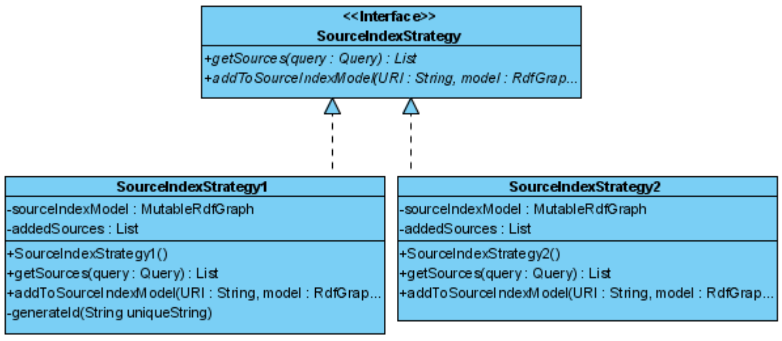
7.4 Source Index Model

A first step to optimize the access to the Environment Model is to maintain a Source Index Model that contains summary information telling us what kind of information is used in a specific source. This Model can be used to determine which of the encountered data sources are relevant for solving a given query. This allows us to no longer consider all the encountered sources when a query is posed to the Environment Model but only the relevant sources. Chapter 4 explained the design details and the two developed strategies: SourceIndexStrategy1, which contains for each source the used predicates and their actual domains and SourceIndexStrategy2, which contains for each source the used predicates. This section explains the implementation details of both the strategies.

A SourceIndexStrategy is a strategy that maintains a Source Index Model. An interface is used to represent the SourceIndexStrategy, to abstract from what kind of information is actually stored in the Source Index Model. Figure 14 shows the class diagram of SourceIndexStrategy and two of its concrete implementations: SourceIndexStrategy1 and SourceIndexStrategy2.

The SourceIndexStrategy interface has two methods:

- GetSources: returns all the relevant sources for a given query. This method firstly analyses the query, extracts the relevant information and constructs a SourceIndexQuery. Secondly this SourceIndexQuery is posed to the Source Index Model, the results of this query are all the relevant sources for the given query. For more details about the followed workflow see sections 5.4, 5.5 and 7.4.3.
- AddToSourceIndexModel: is responsible for extracting the source index information out of the source and adding it to the Source Index Model (see section 5.3).



<<Interface>>
SourceIndexStrategy
+getSources(query : Query) : List
+addToSourceIndexModel(URI : String, model : RdfGrap...
SourceIndexStrategy1
-sourceIndexModel : MutableRdfGraph
-addedSources : List
+SourceIndexStrategy1()
+getSources(query : Query) : List
+addToSourceIndexModel(URI : String, model : RdfGrap...
-generateId(String uniqueString)
SourceIndexStrategy2
-sourceIndexModel : MutableRdfGraph
-addedSources : List
+SourceIndexStrategy2()
+getSources(query : Query) : List
+addToSourceIndexModel(URI : String, model : RdfGrap...

Figure 14: Class diagram SourceIndexStrategy

7.4.1 Source Index Strategy 1

The first concrete implementation is SourceIndexStrategy1, as explained in Chapter 5 this first strategy maintains a Source Index Model that contains information about the used predicates and their corresponding actual domain. This section explains the implementation details of SourceIndexStrategy1.

The state of SourceIndexStrategy1 consists out of two fields:

- SourceIndexModel (MutableRdfGraph): the RDF representation of the SourceIndexModel.
- addedSources (List): a list containing all the sources captured by the Source Index Model, this list allows us to avoid indexing the same source twice.

Next to the two methods inherited of the SourceIndexStrategy, it provides one extra method:

- GenerateID: this method generates a unique id for statements describing the predicate-domain occurrence in a source. For more information why this identifier is necessary see 6.1.2.

7.4.2 Source Index Strategy 2

This section describes the implementation details of SourceIndexStrategy2, which contains a Source Index Model containing information about the used predicates. For more details about the design see sections 5.3.2 and 5.4.2.

The state of SourceIndexStrategy2 consists out of two fields:

- SourceIndexModel (MutableRdfGraph): contains the Source Index Model represented as an RDF Graph.
- addedSources (List): a list containing all the sources of the Source Index Model, this allows us to avoid indexing the same source twice.

7.4.3 Query Analysis

Workflow 1 showed the conceptual process of what happens when a Source Index Query is constructed. This section describes the same process but in terms of the implementation. Figure 15 show the sequence diagram of the first and seconds step of the workflow, i.e., constructing the Abstract Syntax Tree and the SourceIndexQueryTree, while Figure 16 represents the third step, constructing the SourceIndexQuery from the SourceIndexQueryTree.

The implementation of the first step, constructing the Abstract Syntax Tree, is done by using the SPARQLParser class of SPARQL Engine for Java[29] (actions 1 to 3 on Figure 15).

[29] http://sparql.sourceforge.net/

SourceIndexQueryTreeComposer is the component that is responsible for the control flow of constructing the source index query.

In the second step, the AST formed by the SPARQL parser in step one is translated into a SourceIndexQueryTree, an Abstract Syntax Tree representation of the corresponding Source Index Query. The query is constructed in two steps: the where clause is formed by the QueryComposerVisitor, and the select clause by the SourceIndexQueryTreeComposer (action 9 in Figure 8). The reason for this two-phase construction process is the fact that the select variables are not determined by the (original) query itself. In our case the purpose of the Source index Query is selecting the relevant sources, but in other causes it could be something else. To construct the where clause of the Source Index Query, the Visitor pattern offered by the SPARQL Engine API is used; QueryComposerVisitor implements the interface SPARQLVisitor and iterates over the AST (actions 5 to 8).

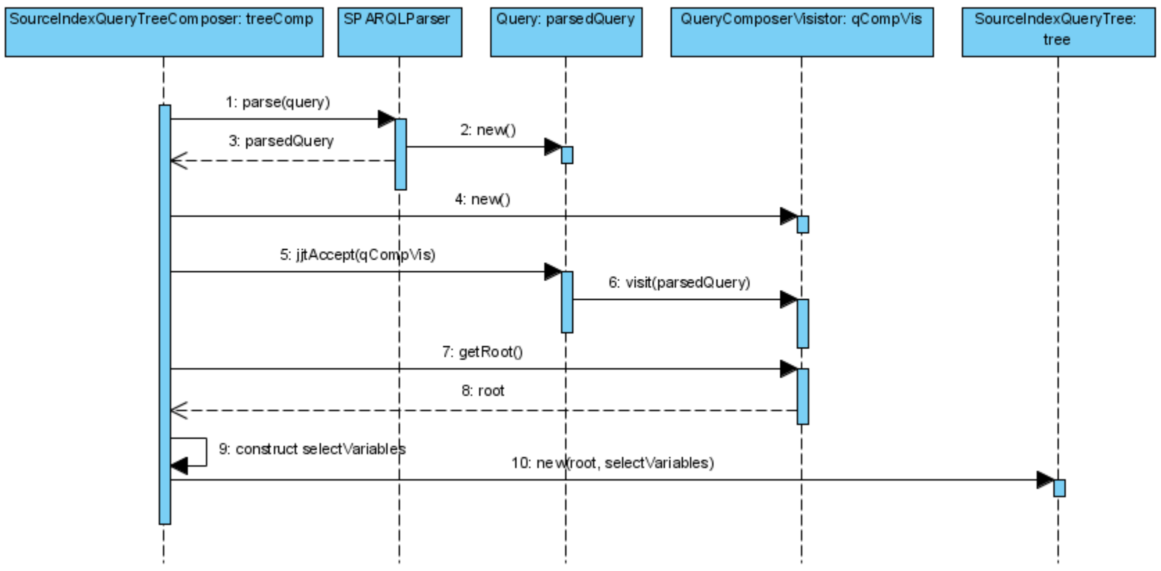

SourceIndexQueryTreeComposer: treeComp
SPARQLParser
Query: parsedQuery
QueryComposerVisistor: qCompVis
SourceIndexQueryTree: tree
1: parse(query)
2: new()
3: parsedQuery
4: new()
5: jjtAccept(qCompVis)
6: visit(parsedQuery)
7: getRoot()
8: root
9: construct selectVariables
10: new(root, selectVariables)

Figure 15: Sequence diagram, constructing SourceIndexQueryTree

The final step constructs the actual source index query (string) from the SourceIndexQueryTree. This is done using a Visitor on the SourceIndexQueryTree. Figure 16 shows the sequence diagram of this process.

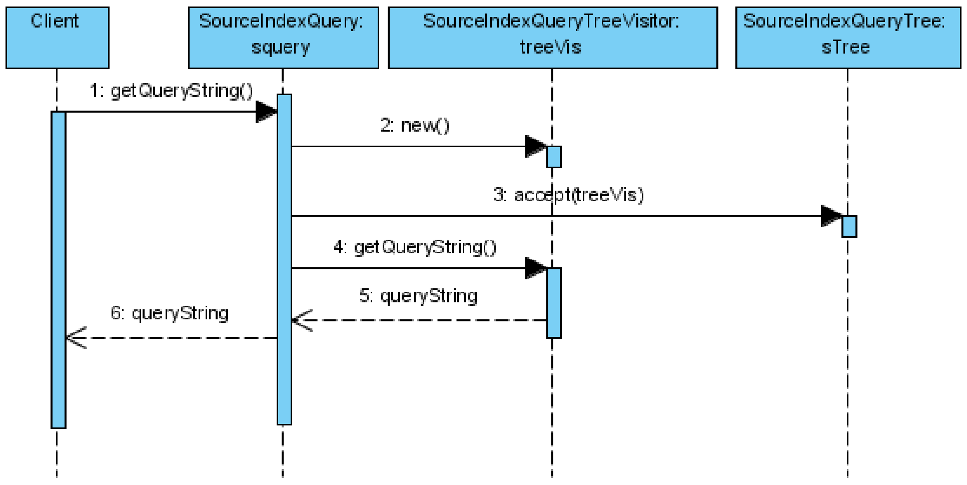


Figure 16: Sequence diagram, constructing SourceIndexQuery

Although several concrete implementation choices have been made, we do want to point out the flexibility of our design. In the following paragraphs, we shortly elaborate on this flexibility, which allows to easily switch to an alternative concrete implementations for most components.

A first observation to make is the fact that evidently, the actual source index query is dependent of the exact format of Source Index Model. We took this consideration into account by providing an abstraction for source index query construction. Currently the source index model is stored as an RDF Graph, but if we decide to store it as an SQL database at a later point in time, our implementation should allow seamlessly "plug in" a component that construct a source index query over this new Source Index model. Therefore, we utilized the visitor pattern, materialized as a SourceIndexQueryTreeVisistor: an interface which can have different concrete implementations. Depending on how the Source Index model is built and accessed, a different concrete implementation of the SourceIndexQueryTreeVisistor interface allows to construct appropriate source index query strings. In line with this design choice, the Source Index Query is an abstract class to allow different types of queries (e.g., SPARQL Query, SQL Query, …). There is a one-to-one mapping between the concrete implementations of SourceIndexQueryTreeVisitors and the subclasses of SourceIndexQuery. The SourceIndexQuery component is responsible for initializing the appropriate SourceIndexQueryTreeVisitor. In our particular setting, were the Source Index Model is stored as an RDF Graph, the Source Index Query is a SPARQLSourceIndexQuery and the SourceIndexQueryTreeVisistor is a SPARQLSourceIndexQueryTreeVisistor.

The second level of abstraction is based on the concrete content of the Source Index Model (i.e., the *type* of information). Depending on which information is stored in the Source Index Model, different information needs to be extracted from the query (e.g., domain and predicate information for SourceIndexStrategy1, only predicate info for SourceIndexStrategy2), and thus a different source index query will be constructed. To allow these variations in Source Indices, we foresaw a SourceIndexQueryTreeComposer interface, with currently two concrete implementations: one for SourceIndexStrategy1 and one for SourceIndexStrategy2. The SourceIndexQueryTreeComposer uses a QueryCompositorVisitor to go over the Abstract Syntax Tree of the given query and extracts the relevant Source Index information. Depending on the type of information that is stored in the Source Index Model, this component needs to extract other kind of information. Therefore the QueryComposerVisitor is an interface with for each SourceIndexStrategy a concrete implementation. The concrete implementation of SourceIndexStrategy1 extracts the used predicates and their domain, while the one of SourceIndexStrategy2 only extracts the used predicates. The Composer is responsible for instantiating the appropriate Visitor.

7.5 Cache

This section covers the implementation of the cache component, a component which is responsible for storing the cache elements and deciding which of them it stores locally by a using a specific eviction strategy. Firstly the abstract Cache interface is discussed; secondly the chosen implementation is describe in a separate subsection.

Cache is implemented as an interface, with currently has three implementations: CacheNone, CacheAll and CacheLRU. CacheNone caches no information and is implemented to simulate the behavior of an Environment Manager which does not store anything of the encountered data sources locally; it will thus download all information every time it is needed. This may be necessary for mobile devices with very limited storage capacity. CacheAll makes it possible to cache all of the discovered information, this cache is implemented for evaluation reason, as it is unrealistic in a real world setting where storage capacity is limited and there is simply not enough space to cache all the encountered data sources. CacheLRU is a cache implementation with as eviction strategy Least Recently Used as explained in section 6.2.

Figure 17 shows the class diagram of Cache and the relevant components. A cache stores certain elements, called cache elements which are an abstract representation of the elements that can be stored in the cache. In section 6.1 we discussed what kind of information we will cache: the entire encountered source, stored as an RDF Graph, or the triples using a specific predicate, stored as a list structure. Therefore we have two concrete implementations of cache elements: CacheElementRdfGrpah, which contains an RDF Graph and is used in to cache entire sources and CacheElementPredicate, which contains the list structure containing the triples using a specific predicate and is used to cache triples of a specific predicate. Every concrete implementation of cache element provides access to its value and its size (calculated in bytes).

The Cache interface provides access to its cache elements via four operations:

- Contains: indicates whether or not the cache contains a cache element with the specific key.
- Get: returns the value for the particular key, in case of CacheElementRdfGraph the value will be an RdfGraph while in case of CacheElementPredicate, the value will be a list of TripleElements. A triple element is an abstract representation which contains information about a triple using the predicate: it contains the subject, subject type, object, object type and source in which the triple occurs. See section 6.1.2 for more information.
- Add: adds a cache element to the cache.
- Update: changes the cache element of a particular key to the new cache element.

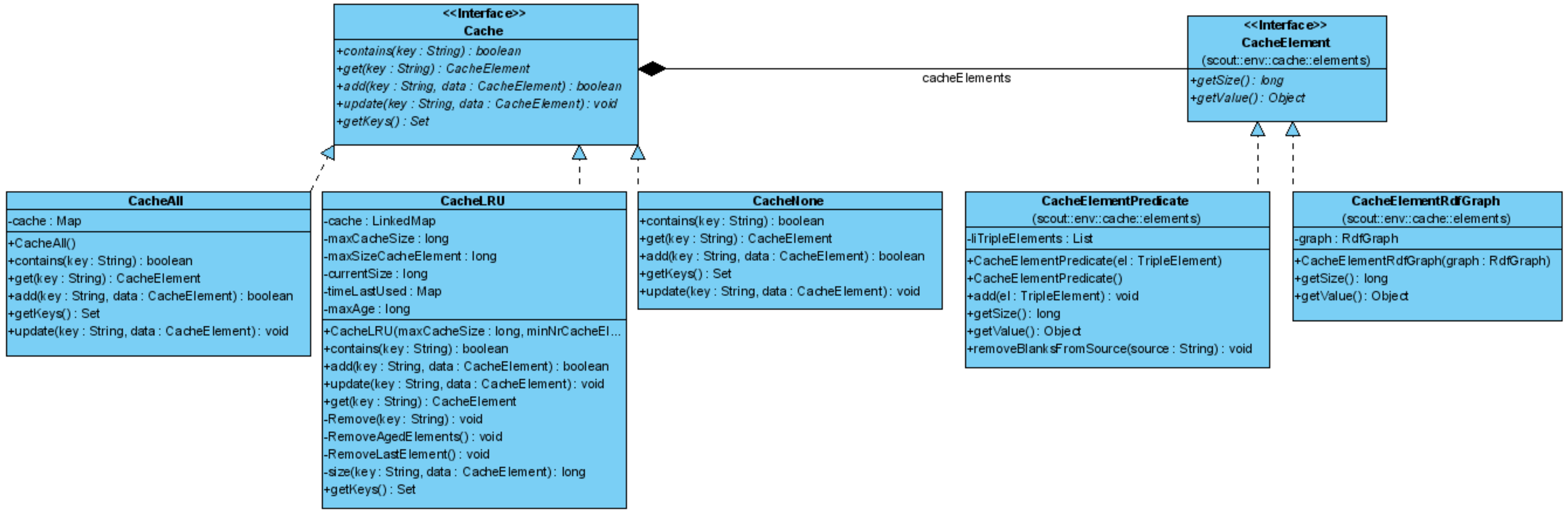

<<Interface>>
Cache
+contains(key : String) : boolean
+get(key : String) : CacheElement
+add(key : String, data : CacheElement) : boolean
+update(key : String, data : CacheElement) : void
+getKeys() : Set
cacheElements
<<Interface>>
CacheElement
(scout::env::cache::elements)
+getSize() : long
+getValue() : Object
CacheAll
-cache : Map
+CacheAll()
+contains(key : String) : boolean
+get(key : String) : CacheElement
+add(key : String, data : CacheElement) : boolean
+getKeys() : Set
+update(key : String, data : CacheElement) : void
CacheLRU
-cache : LinkedMap
-maxCacheSize : long
-maxSizeCacheElement : long
-currentSize : long
-timeLastUsed : Map
-maxAge : long
+CacheLRU(maxCacheSize : long, minNrCacheEl...
+contains(key : String) : boolean
+add(key : String, data : CacheElement) : boolean
+update(key : String, data : CacheElement) : void
+get(key : String) : CacheElement
-Remove(key : String) : void
-RemoveAgedElements() : void
-RemoveLastElement() : void
-size(key : String, data : CacheElement) : long
+getKeys() : Set
CacheNone
+contains(key : String) : boolean
+get(key : String) : CacheElement
+add(key : String, data : CacheElement) : boolean
+getKeys() : Set
+update(key : String, data : CacheElement) : void
CacheElementPredicate
(scout::env::cache::elements)
-liTripleElements : List
+CacheElementPredicate(el : TripleElement)
+CacheElementPredicate()
+add(el : TripleElement) : void
+getSize() : long
+getValue() : Object
+removeBlanksFromSource(source : String) : void
CacheElementRdfGraph
(scout::env::cache::elements)
-graph : RdfGraph
+CacheElementRdfGraph(graph : RdfGraph)
+getSize() : long
+getValue() : Object

Figure 17: Cache Class diagram

7.5.1 Least Recently Used

The previous paragraphs explained the implementation details of the Cache interface; the following paragraphs discuss the implementation details of CacheLRU, a cache implementation with as eviction strategy Least Recently Used as explained in section 6.2. The cache elements that are stored in CacheLRU are maintained in a new data structure called LinkedMap, which is a LinkedList and a Map at the same time. It has as benefit that it allows determining the first and last element, and moving the elements of position in O(1) while the elements are key-value pairs.

CacheLRU has six fields:

- cache (LinkedMap): a data structure to store the cache elements.
- maxCacheSize (long): variable indicating what the maximum cache size is.
- maxSizeCacheElement (long): variable indicating the maximum size of a cache, which is computed by dividing the maxCacheSize by the minimum number of elements that should be stored in the cache.
- currentSize (long): the current size of the cache.
- timeLastUsed (Map): a map containing a timestamp for all the cached elements indicating when it was last requested, to allow us to determine what elements are unused for too long and thus may be evicted from the cache.
- maxAge (long): the maximum time in milliseconds that a cache element may remain unused in the cache.

CacheLRU has the following methods:

- RemoveAgedElement: removes all the cache elements that are unused for too long, in other words, evicts all the elements that have exceeded the maxAge. The cost of this operation is O(n) since for all the elements in the cache, it needs to be checked whether or not the element exceeded the maxAge. Optimizing this algorithm is considered to be future work.
- RemoveLastElement: removes the least recently used cache element, with as cost O(1).
- Size: calculates the size of a cache element by considering the size of the key (IRI) and the cache element.
- Add: adds a cache element to the cache, while doing so, it adss a timestamp for the element to the map timeLastUsed as well. For more details about the algorithm see section 6.2.3.
- Update: is responsible for updating the cache element. For more details about the algorithm see section 6.2.2.

- Get: returns the corresponding cache element, more details about the algorithm can be found in section 6.2.1.

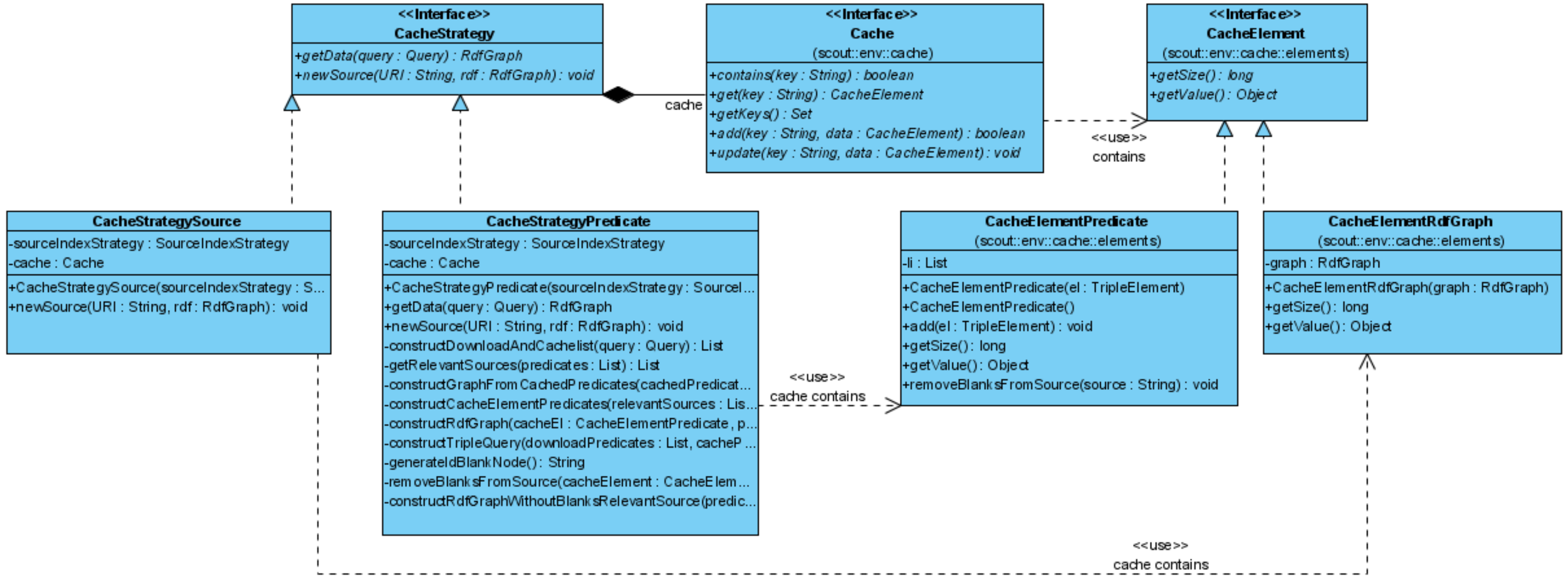
7.6 Cache Strategy

This section covers the implementation details of the cache strategy component, which is responsible for providing all the relevant information for a specific query, for maintaining a cache to store the encountered information and for deciding what type of information (i.e., entire encountered source or triples using a specific predicate) is stored in the cache. Figure 18 shows a class diagram of cache strategy and all the relevant components.

Cache Strategy is implemented as a Java interface with currently two concrete implementations: CacheStrategySource, which represents the caching mechanism where entire encountered sources are cached, and CacheStrategyPredicate, which represents the caching mechanism where triples using a specific predicates are cached. Depending on the used cache strategy, different types of cache elements are stored in the cache, CacheStrategySource stores CacheElementRdfGraph objects while CacheStrategyPredicate stores CacheElementPredicate objects.

The interface, called CacheStrategy, has two operations:

- GetData: returns all relevant encountered RDF information (i.e., RDF triples) for a particular query combined into one RdfGraph.
- NewSource: deals with newly encountered data sources. Does information of the new source needs to be added to the cache or not?



<<Interface>>
CacheStrategy
+getData(query : Query) : RdfGraph
+newSource(URI : String, rdf : RdfGraph) : void
<<Interface>>
Cache
(scout::env::cache)
+contains(key : String) : boolean
+get(key : String) : CacheElement
+getKeys() : Set
+add(key : String, data : CacheElement) : boolean
+update(key : String, data : CacheElement) : void
cache
<<Interface>>
CacheElement
(scout::env::cache::elements)
+getSize() : long
+getValue() : Object
<<use>>
contains
CacheStrategySource
-sourceIndexStrategy : SourceIndexStrategy
-cache : Cache
+CacheStrategySource(sourceIndexStrategy : S...
+newSource(URI : String, rdf : RdfGraph) : void
CacheStrategyPredicate
-sourceIndexStrategy : SourceIndexStrategy
-cache : Cache
+CacheStrategyPredicate(sourceIndexStrategy : SourceI...
+getData(query : Query) : RdfGraph
+newSource(URI : String, rdf : RdfGraph) : void
-constructDownloadAndCachelist(query : Query) : List
-getRelevantSources(predicates : List) : List
-constructGraphFromCachedPredicates(cachedPredicat...
-constructCacheElementPredicates(relevantSources : Lis...
-constructRdfGraph(cacheEl : CacheElementPredicate, p...
-constructTripleQuery(downloadPredicates : List, cacheP ...
-generateIdBlankNode() : String
-removeBlanksFromSource(cacheElement : CacheElem...
-constructRdfGraphWithoutBlanksRelevantSource(predic...
<<use>>
cache contains
CacheElementPredicate
(scout::env::cache::elements)
-li : List
+CacheElementPredicate(el : TripleElement)
+CacheElementPredicate()
+add(el : TripleElement) : void
+getSize() : long
+getValue() : Object
+removeBlanksFromSource(source : String) : void
CacheElementRdfGraph
(scout::env::cache::elements)
-graph : RdfGraph
+CacheElementRdfGraph(graph : RdfGraph)
+getSize() : long
+getValue() : Object
<<use>>
cache contains

Figure 18: Class diagram CacheStrategy

7.6.1 Caching entire sources

The previous paragraphs explained the concrete implementation details of the CacheStrategy interface, the following paragraphs explain the implementation details of CacheStrategySource a concrete implementation of CacheStrategy. It maintains a cache where the cache elements contain the full RDF graph of one particular source, i.e., the cache granularity is an (encountered) source. The RDF information of one specific source is maintained in a CacheElementRdfGraph, a concrete implementation of Cache Element that was discussed in section 7.5.

The CacheStrategySource has two fields:

- Cache (Cache): a cache to store (some of) the sources.
- SourceIndexStrategy (SourceIndexStrategy): a source index strategy to determine the relevant sources for a specific query (for more details see section 7.3).

Notice that the cache strategy itself is not responsible for maintaining the source index strategy; it only uses it to delegate the task of determining the relevant sources for a particular query.

Cache Strategy Source provides an implementation for the two methods of the Cache Strategy interface:

- GetData: retrieves all the information that is or might be relevant to solve the particular query. For the details about the followed algorithm see section 6.1.1.
- NewSource: performs the necessary actions whenever a new source is encountered. In this particular case, this method does not perform any actions as explained in section 6.1.1.

7.6.2 Caching triples using a specific predicate

The previous paragraphs explained the implementation details of CacheStrategySource. This particular cache strategy stored the encountered sources entirely in the cache. As explained in section 6.1.2, the second cache strategy stores triples using a specific predicate in the cache. First of all the global idea was given, followed by the complicating factors and finally the final variant. This section only discusses the implementation details of the final variant, since only this correct variant is implemented.

This implementation consists of a cache where the cache elements contain all the triples using a specific predicate. It therefore used CacheElementPredicate, a subclass of the CacheElement, which implements the generic CacheElement interface (see section 7.5 for more details about the cache element).

The CacheStrategyPredicate has two fields:

- Cache (Cache): a cache to store (some of) the sources.
- SourceIndexStrategy (SourceIndexStrategy): a source index strategy to determine the relevant sources for a specific query (for more details see section 7.3).

Notice that the Cache Strategy itself is not responsible for maintaining the source index strategy; it only uses it to delegate the task of determining the relevant sources for a particular query.

Cache Strategy Predicate provides an implementation for the two methods of the Cache Strategy interface:

- GetData: retrieves all the information that is or might be relevant to solve the particular query. For details about the algorithm to determine which information is relevant see section 6.1.2.
- NewSource: performs the necessary actions whenever a new source is encountered. In this particular case, this method updates the cached predicates with the triples of the newly encountered source which use the specific predicates. For more information see section 6.1.2.

Apart from these two methods, Cache Strategy Predicate has nine other private methods each responsible for a specific part of the workflow (Workflow 4) shown in section 6.1.2:

- ConstructDownloadAndCachelist: constructs two lists, one containing the cached and another containing the uncached predicates relevant for the query. (The last paragraph provides some more information on this topic).
- GetRelevantSources: retrieves the sources relevant for the uncached predicates by consulting the Source Index Strategy. Firstly it constructs a query using each one of the predicates. By constructing this query, we can ask the SourceIndexStrategy to determine the relevant sources in a similar as when determining the relevant sources for a query posed to the Environment Model.
- ConstructGraphFromCachedPredicates: constructs an RDF Graph containing all the cached triples, the triples using a blank node from one of the relevant sources are not included in this graph.
- ConstructCacheElementPredicates: constructs an RDF Graph for each of the predicates by downloading the relevant sources and extracting the triples that use one of the specific predicates.
- ConstructRdfGraph: construct an RDF Graph out of a CacheElementPredicate and the corresponding predicate. The CacheElementPredicate object contains information about all the triples using the specific predicate. The method constructs an RDF Graph that contains all triples using the specific predicate from the CacheElementPredicate objects.

- ConstructTripleQuery: constructs a query that selects all the triples using a non-cached predicate or a cached predicates using a blank node. The latter is necessary to be able to ensure that the blank node has the appropriate identifier. See 6.1.2 for more details.
- GenerateIdBlankNode: generates a unique id for the blank node.
- RemoveBlanksFromSource: extracts all the triples of the CacheElementPredicate that do not use a blank node from a relevant source. This method returns a CacheElementPredicate.
- ConstructRdfGraphWithoutBlanksRelevantSource: is the method that constructs an RDF Graph for the CacheElementPredicate that excludes all the triples using a blank node from a relevant source. The difference with the previous method is that this method returns an RDF Graph while the previous one returned a CacheElementPredicate.

Section 6.1.2 explained the concept of the caching of triples using specific predicates. The first step in Workflow 4 is to construct a predicate list, a list of all the predicates that are used in the query. The constructing of this predicate list is offered as a remote service since currently the used API (SPARQL Engine for Java) is not available for Java ME. Figure 19 shows the class diagram of the involved components; Figure 20 shows the sequence diagram for constructing the predicate list of a query. The main component is the PredicateSetComposer, which interacts with the client (i.e., the SCOUT framework running on a mobile devices), and uses SPARQLParser for building the Abstract Syntax Tree (AST) from a SPARQL query. After composing the AST, the PredicateQueryVisitor is used to extract the predicates from the AST and collect them in a Set.

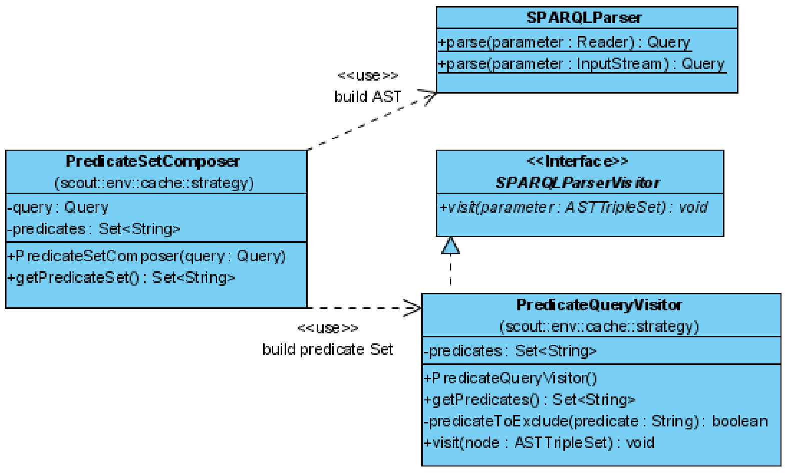

SPARQLParser
+parse(parameter : Reader) : Query
+parse(parameter : InputStream) : Query
<<use>>
build AST
PredicateSetComposer
(scout::env::cache::strategy)
-query : Query
-predicates : Set<String>
+PredicateSetComposer(query : Query)
+getPredicateSet() : Set<String>
<<Interface>>
SPARQLParserVisitor
+visit(parameter : ASTTripleSet) : void
<<use>>
build predicate Set
PredicateQueryVisitor
(scout::env::cache::strategy)
-predicates : Set<String>
+PredicateQueryVisitor()
+getPredicates() : Set<String>
-predicateToExclude(predicate : String) : boolean
+visit(node : ASTTripleSet) : void

Figure 19: Class diagram PredicateSetComposer

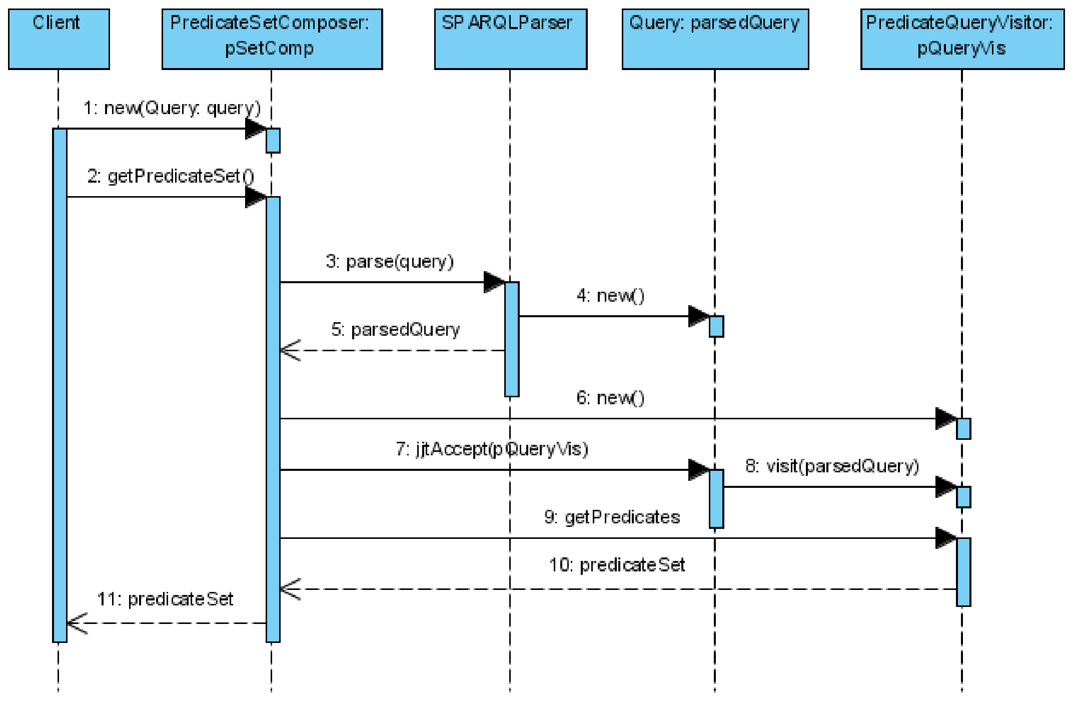


Figure 20: Sequence diagram Construct predicate set

7.7 Used Design Patterns

This last section of the implementation chapter summarizes the design patterns that were used to develop the efficient query mechanism for the Environment Model. The design patterns themselves are not explained (we refer to [26] for details on the design patterns that were used); however, the reason for choosing the particular design patterns is explained here. We used five design patterns in our work: singleton, observer, strategy, composite and visitor. The following paragraphs each explain one design pattern.

The Singleton Pattern is used for the Environment Manager class. As there is only one Environment Model to manage, there should only be one unique instance and thus the singleton pattern is a logical choice here to provide one single instance (and point of access).

The Observer Pattern is used between the Relation Manager class and the Environment Manager class. The Relation Manager should notify its dependents whenever its state changes, the Relation Manager plays the role of Observable and the Environment Manager the one of Observer. Figure 21: Class diagram Observer Pattern, Relation Manager and Environment ManagerFigure 21 shows the class diagram of the involved components.

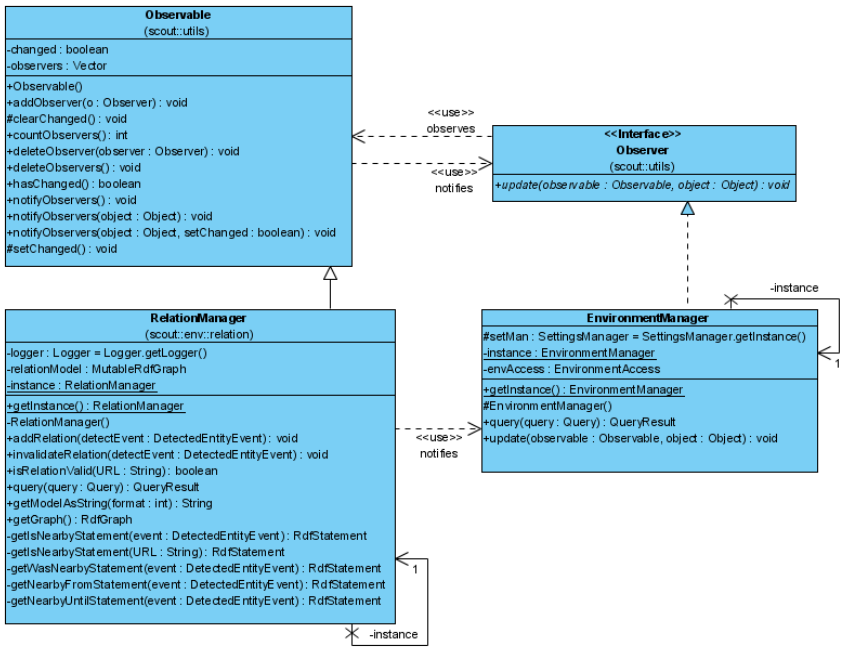
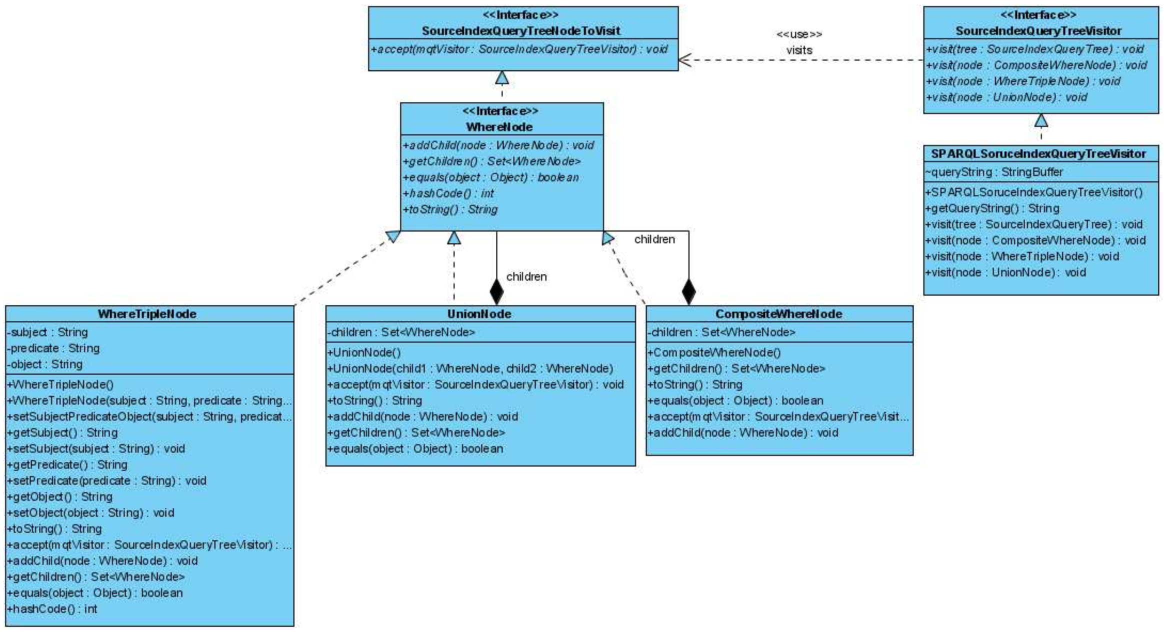

Observable
(scout::utils)
-changed : boolean
-observers : Vector
+Observable()
+addObserver(o : Observer) : void
#clearChanged() : void
+countObservers() : int
+deleteObserver(observer : Observer) : void
+deleteObservers() : void
+hasChanged() : boolean
+notifyObservers() : void
+notifyObservers(object : Object) : void
+notifyObservers(object : Object, setChanged : boolean) : void
#setChanged() : void
<<use>>
observes
<<use>>
notifies
<<Interface>>
Observer
(scout::utils)
+update(observable : Observable, object : Object) : void
-instance
1
RelationManager
(scout::env::relation)
-logger : Logger = Logger.getLogger()
-relationModel : MutableRdfGraph
-instance : RelationManager
+getInstance() : RelationManager
-RelationManager()
+addRelation(detectEvent : DetectedEntityEvent) : void
+invalidateRelation(detectEvent : DetectedEntityEvent) : void
+isRelationValid(URL : String) : boolean
+query(query : Query) : QueryResult
+getModelAsString(format : int) : String
+getGraph() : RdfGraph
-getIsNearbyStatement(event : DetectedEntityEvent) : RdfStatement
-getIsNearbyStatement(URL : String) : RdfStatement
-getWasNearbyStatement(event : DetectedEntityEvent) : RdfStatement
-getNearbyFromStatement(event : DetectedEntityEvent) : RdfStatement
-getNearbyUntilStatement(event : DetectedEntityEvent) : RdfStatement
<<use>>
notifies
EnvironmentManager
#setMan : SettingsManager = SettingsManager.getInstance()
-instance : EnvironmentManager
-envAccess : EnvironmentAccess
+getInstance() : EnvironmentManager
#EnvironmentManager()
+query(query : Query) : QueryResult
+update(observable : Observable, object : Object) : void
1
-instance

Figure 21: Class diagram Observer Pattern, Relation Manager and Environment Manager

The Strategy Pattern is used two times. A first time it is used to encapsulate the type of information that is stored in the cache (the CacheStrategy class). Currently, there are two CacheStrategies, which are easily interchangeable due to the use of the Strategy pattern. Another benefit is that the CacheStrategy can change at runtime. The second use of the Strategy Pattern is materializes in the context of the Cache. Here it is deployed to encapsulate the algorithm which determines what elements can stay in the cache and which cannot, in other words, to encapsulate the eviction strategy. Again, the benefit is that the eviction strategy can change at runtime, and additionally the complex eviction algorithm is not exposed to the clients (CacheStrategy) using it. Currently there are three concrete Cache implementations (CacheAll, CacheNone, CacheLRU). Other eviction strategy can be implemented and easily added to the system. Figure 18 show the class diagram of the involved components.

The composite pattern is used to represent the whereTree of a SourceIndexQuery. It composes the objects into a tree-like structure and allows components to have none, one or more children while still addressing the components in a similar way. By using the composite pattern to form a tree structure, any operations on this tree structure are easily implemented using the Visitor Pattern. The class diagram is shown in Figure 22.



<<Interface>>
SourceIndexQueryTreeNodeToVisit
+accept(mqtVisitor : SourceIndexQueryTreeVisitor) : void
<<use>>
visits
<<Interface>>
SourceIndexQueryTreeVisitor
+visit(tree : SourceIndexQueryTree) : void
+visit(node : CompositeWhereNode) : void
+visit(node : WhereTripleNode) : void
+visit(node : UnionNode) : void
<<Interface>>
WhereNode
+addChild(node : WhereNode) : void
+getChildren() : Set<WhereNode>
+equals(object : Object) : boolean
+hashCode() : int
+toString() : String
SPARQLSoruceIndexQueryTreeVisitor
~queryString : StringBuffer
+SPARQLSoruceIndexQueryTreeVisitor()
+getQueryString() : String
+visit(tree : SourceIndexQueryTree) : void
+visit(node : CompositeWhereNode) : void
+visit(node : WhereTripleNode) : void
+visit(node : UnionNode) : void
children
children
WhereTripleNode
-subject : String
-predicate : String
-object : String
+WhereTripleNode()
+WhereTripleNode(subject : String, predicate : String...
+setSubjectPredicateObject(subject : String, predicat...
+getSubject() : String
+setSubject(subject : String) : void
+getPredicate() : String
+setPredicate(predicate : String) : void
+getObject() : String
+setObject(object : String) : void
+toString() : String
+accept(mqtVisitor : SourceIndexQueryTreeVisitor) : ...
+addChild(node : WhereNode) : void
+getChildren() : Set<WhereNode>
+equals(object : Object) : boolean
+hashCode() : int
UnionNode
-children : Set<WhereNode>
+UnionNode()
+UnionNode(child1 : WhereNode, child2 : WhereNode)
+accept(mqtVisitor : SourceIndexQueryTreeVisitor) : void
+toString() : String
+addChild(node : WhereNode) : void
+getChildren() : Set<WhereNode>
+equals(object : Object) : boolean
CompositeWhereNode
-children : Set<WhereNode>
+CompositeWhereNode()
+getChildren() : Set<WhereNode>
+toString() : String
+equals(object : Object) : boolean
+accept(mqtVisitor : SourceIndexQueryTreeVisit...
+addChild(node : WhereNode) : void

Figure 22: Class diagram Composite and Visitors pattern

The final pattern, which is used throughout the implementation, is the Visitor pattern. It is used to construct a predicate list of a query with help of the SPARQL Engine; to construct a whereTree out of the Abstract Syntax Tree; and finally, to construct a query string out of a SourceIndexQueryTree (shown in Figure 22). In each case, the Visitor pattern is used to perform an operation on elements of an object structure.

Chapter 8 Evaluation

Until now, the Environment Model, a Model which comprises metadata on physical entities found nearby the user and the user's own profile information, has been stored as a fully materialized view; in other words, all of the data (i.e., encountered data sources) is kept locally.

This thesis investigated how the Environment Model can be constructed and managed more efficiently. To fulfill this purpose, we have investigated several ways of storing source index information on encountered data sources to be able to determine which sources contain relevant information for a given query. Secondly, we have developed several caching strategies where some data from encountered sources is kept locally, to avoid having to download relevant sources (as identified by one of the strategies mentioned above) every time they are needed to solve a query posed to the Environment Model.

This chapter is dedicated to the evaluation and proof of concept of the two developed mechanisms. The first section describes the test environment, followed by two sections discussing an evaluating the test results of the developed source index strategies and cache strategies, respectively.

8.1 Testing environment

The test environment consists of a client part, i.e., the SCOUT framework, and a server part, which is deployed on a pc and responsible for resolving queries and forming Source Index Queries. The client is run on the Sun Java™ Wireless Toolkit for CLDC emulator via Netbeans IDE 6.8 on a HP Compaq 6710b laptop, which has an Intel ® Core™ 2 Duo CPU T9300 2.50 Ghz processor, 4GB RAM and Windows 7 64-bit version as operation system. The server is deployed on a Tomcat v6.0 server on the same laptop. Ideally, the client would have run on a mobile device; however, the measurement of execution times was much less time consuming using a simulator.

Every test case is executed 100 times spread over several days. All the results mentioned in this chapter are the average of those 100 runs. These test cases use fifty RDF sources spread over four servers. The distribution for the source index strategy is 12 sources on wilma.vub.ac.be, 13 on clinicadentalnadal.com, 13 on belgischebieren.eu and 12 on vwoensel.com. For the cache strategy it is slightly different 11 sources on wilma.vub.ac.be, 13 on pvanwoenssel.be, 13 on clinicadentalnadal.com and 13 on belgischebieren.eu. The domains vwoensel.com en pvanwoensel.be are deployed on the same server.

8.2 Source Index Strategy

This section evaluates two strategies (called source index strategies) which we have developed to store summary information on encountered data sources in a so-called Source Index Model. This evaluation takes into account the following criteria: time needed to construct the Source Index Model; size of the Source Index Model; execution time of queries on the Source Index Model (i.e., time needed to determine relevant sources); number of relevant sources found in the Source Index Model for a given query; and time needed to execute the original query on the found relevant sources. The first section provides details on the employed test cases, followed by sections evaluating the strategies with regards to the criteria above. The last section provides conclusions that can be drawn from the evaluation of the strategies.

In our first strategy, SourceIndexStrategy1, information on the predicates and their domains found in encountered data sources is kept in the Source Index Model, while in SourceIndexStrategy2 only information on the predicates is kept. In addition, we consider a worst-case strategy where no summary information is kept (i.e., all data sources are considered relevant) called SourceIndexStrategyNone, and a best-case strategy where perfect source index information is available on each data source (i.e., only data sources that are actually relevant are included), called SourceIndexStrategyBest. The latter two are thus hypothetical strategies, employed to compare the two developed strategies to worst- and best-case scenarios.

8.2.1 The test cases

In order to evaluate the source index strategies, seven test cases are used, numbered from 1 to 7. The same seven test cases are employed for the evaluation of each of the aforementioned criteria. Test cases 1 to 4 have a Source Index Model containing source index information on one data source, test 5 has a Source Index Model summarizing two similar data sources (actually, the sources from test case 1 and 2), test 6 has a containing source index information on two similar sources (again, the sources from test case 1 and 2) and two very different files (actually, the from test case 3 and 4) and finally test 7 has a Model summarizing 50 files, some of which are similar and some not.

The test cases for the first two evaluation criteria (i.e., Source Index Model construction time and size) involve constructing the Source Index Model. For the first criterion, the time in milliseconds necessary to form the Source Index Model is measured, while for the second criterion the size of the constructed model in bytes is measured. Note that these criteria are not relevant for the worst- and best-case strategy mentioned before, as these strategies do not involve constructing a Source Index Model.

The tests for the third and fourth test criteria (i.e., the time necessary to determine the relevant sources, and the number of relevant sources found) involve querying the constructed Source

Index Models for the relevant sources given a certain query, and measuring the query execution time (in ms) and the number of sources considered relevant. Three given queries are considered: one with a low complexity, one with medium complexity, and one with high complexity. A low complexity query is a basic query consisting of two triple patterns. A medium complexity query is composed of five to six triple patterns. A query containing unions, filter clauses and a total of ten to twelve triple patterns is considered to be a high complexity query. For more information about the used queries, see section 8.2.1.1. Again, note that for the third criterion, the best- and worst-case strategy are irrelevant, as in the former case the relevant sources are determined manually, and in the latter case no selection of relevant sources takes place.

The test for the sixth and last criterion (i.e., time necessary to execute the original query on the relevant sources) executes the given queries (as defined above) over the relevant sources obtained from the previous tests, and measures the query execution time.

8.2.1.1 Test queries

This subsection describes the "given" queries that are used in the test cases for the last three test criteria. As mentioned above, the complexity of the queries varies from low to high; this allows us to see how much influence the query complexity has on the time necessary to determine the relevant sources. For each query the goal, complexity and numbers of tests in which the query is used is mentioned.

This first query has a low complexity, and selects the title of a music record. It is used in tests 1, 2, 5 and 6.

```
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?title
WHERE
{
   ?r a mo:Record .
   ?r dc:title ?title .
}
```

SPARQL query 12: Low complexity for tests 1, 2, 5 and 6

The following query is used for tests 1, 5 and 6 and has a medium complexity. It lists the titles of the records and tracks of the group 'Absynthe Minded'.

```
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX foaf: <http://xmlns.com/foaf/spec/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?record ?track
WHERE
{
   ?g a mo:MusicGroup .
   ?g foaf:name 'Absynthe Minded' .
   ?r a mo:Record .
   ?r dc:creator ?g .
   ?r dc:title ?record .
   ?r mo:has_track ?t .
   ?t a mo:Track .
   ?t dc:title ?track .
}
```

SPARQL query 13: Medium complexity query for tests 1, 5 and 6

The next query is a variation of the previous one: this time the records and tracks of the group ‘Florence And The Machine’ are listed. This query is used for test 2 and has a medium complexity.

```
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX foaf: <http://xmlns.com/foaf/spec/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?record ?track
WHERE
{
   ?g a mo:MusicGroup .
   ?g foaf:name 'Florence And The Machine' .
   ?r a mo:Record .
   ?r dc:creator ?g .
   ?r dc:title ?record .
   ?r mo:has_track ?t .
   ?t a mo:Track .
   ?t dc:title ?track .
}
```

SPARQL query 14: Medium complexity query for test 2

This query lists the names of the music groups starting with ‘a’ or ‘b’ (‘a’ and ‘b’ are case insensitive), along with their record and track titles. The listing is ordered ascending first by name, then by record title and finally by track title. It is used for tests 1, 2, 5 and 6 and has a high complexity.

```
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX foaf: <http://xmlns.com/foaf/spec/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?name ?record ?track
WHERE
{
   {
      ?g a mo:MusicGroup .
      ?g foaf:name ?name .
      ?r a mo:Record .
      ?r dc:creator ?g .
      ?r dc:title ?record .
      ?r mo:has_track ?t .
      ?t a mo:Track .
      ?t dc:title ?track .
      FILTER(regex(str(?name) ,'^a',  'i'))
   }
   UNION
   {
      ?g a mo:MusicGroup .
      ?g foaf:name ?name .
      ?r a mo:Record .
      ?r dc:creator ?g .
      ?r dc:title ?record .
      ?r mo:has_track ?t .
      ?t a mo:Track .
      ?t dc:title ?track .
      FILTER(regex(str(?name) ,'^b',  'i'))
   }
}
ORDER BY ?name ?record ?track
```

SPARQL query 15: High complexity query for tests 1, 2, 5 and 6

The next query is a low complexity query used for test 3 that lists all the University labs along with the employees and their function.

```
PREFIX region: <http://wise.vub.ac.be/region/>
SELECT ?l ?w ?function
WHERE
{
      ?l a region:University-Lab .
      ?w region:works-in-unit ?l .
      ?w a ?function .
}
```

SPARQL query 16: Low complexity query for test 3

In the medium complexity query below, the departments along with their labs, the labs employees and their function are listed. As an extra condition, the employee must be a PhD. This query is used for test 3.

```
PREFIX region: <http://wise.vub.ac.be/region/>
SELECT ?d ?l ?w ?function
WHERE
{
   ?d a region:University-Department .
   ?d region:has-academic-unit ?l .
   ?l a region:University-Lab .
   ?w region:works-in-unit ?l .
   ?w a ?function .
   ?w region:has-academic-degree <http://www.aktors.org/ontology/portal#PhD> .
}
```

SPARQL query 17: Medium complexity query for test 3

This high complexity query is a query that lists the PhD employees along with their function, but only those that work for the lab 'http://wise.vub.ac.be' and 'http://soft.vub.ac.be' are shown. This query is used for test 3.

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

```
SELECT ?l ?w ?function
WHERE
{
 {
      ?d a region:University-Department .
      ?d region:has-academic-unit ?l .
      ?l a region:University-Lab .
      ?w region:works-in-unit ?l .
      ?w a ?function .
      ?w region:has-academic-degree <http://www.aktors.org/ontology/portal#PhD> .
      FILTER(str(?l) = 'http://wise.vub.ac.be/')
 }
 UNION
 {
      ?d a region:University-Department .
      ?d region:has-academic-unit ?l .
      ?l a region:University-Lab .
      ?w region:works-in-unit ?l .
      ?w a ?function .
      ?w region:has-academic-degree <http://www.aktors.org/ontology/portal#PhD> .
      FILTER(str(?l) = 'http://soft.vub.ac.be/')
   }
 }
```

SPARQL query 18: High complexity query for test 3

Query number 8 selects all the names and birthdays, has a low complexity and it is used for tests 4 and 7.

```
PREFIX foaf: <http://xmlns.com/foaf/spec/>
PREFIX   dc: <http://purl.org/dc/terms/>
SELECT ?name ?day
WHERE
{
   ?a dc:title ?name .
   ?a foaf:birthday ?day .
}
```

SPARQL query 19: Low complexity query for tests 4 and 7

Next to the name and birthday, the following query also selects the interests of the person. As an extra condition, the person must have as gender ‘male’. This medium complexity query is used for tests 4 and 7.

```
PREFIX foaf: <http://xmlns.com/foaf/spec/>
PREFIX   dc: <http://purl.org/dc/terms/>
SELECT ?name ?day ?interest
WHERE
{
   ?p1 dc:title ?name .
   ?p1 foaf:birthday ?day .
   ?p1 foaf:gender     'male' .
   ?p1 foaf:interest ?b1 .
   ?b1 foaf:topic ?interest .
}
```

SPARQL query 20: Medium complexity query for tests 4 and 7

This last high complexity query selects the name, birthday, gender and interest of the persons with as gender ‘male’ or ‘female’. The query results are ordered ascending by name. This query is used for tests 4 and 7

```
PREFIX foaf: <http://xmlns.com/foaf/spec/>
PREFIX dc: <http://purl.org/dc/terms/>
SELECT ?name ?day ?gender ?interest
WHERE
{
   {
      ?p1 dc:title ?name .
      ?p1 foaf:birthday ?day .
      ?p1 foaf:gender  ?gender .
      ?p1 foaf:interest ?b1 .
      ?b1 foaf:topic ?interest .
      FILTER
      (
         str(?gender) = 'male'
      )
   }
   UNION
   {
      ?p1 dc:title ?name .
      ?p1 foaf:birthday ?day .
      ?p1 foaf:gender  ?gender .
      ?p1 foaf:interest ?b1 .
      ?b1 foaf:topic ?interest .
      FILTER
      (
         str(?gender) = 'female'
      )
   }
}
ORDER BY ?name
```

SPARQL query 21: High complexity query for tests 4 and 7

8.2.2 Criterion 1: Construction Time

This section compares SourceIndexStrategy1 and SourceIndexStrategy2, based on the time needed to extract the source index information from the sources and construct the Source Index Model. Chart 1 shows the test results for the seven test cases. As mentioned in the introduction, the Source Index Model contains one file for tests 1-4, two files for test 5, four files for test 6 and fifty files for test 7. These tests were not executed for the best- and worst case strategies, as these do not involve constructing Source Index Models (idem for the following 2 criteria). The horizontal axis represents the test number, while the vertical axis represents the time necessary to construct the Source Index Model in milliseconds.

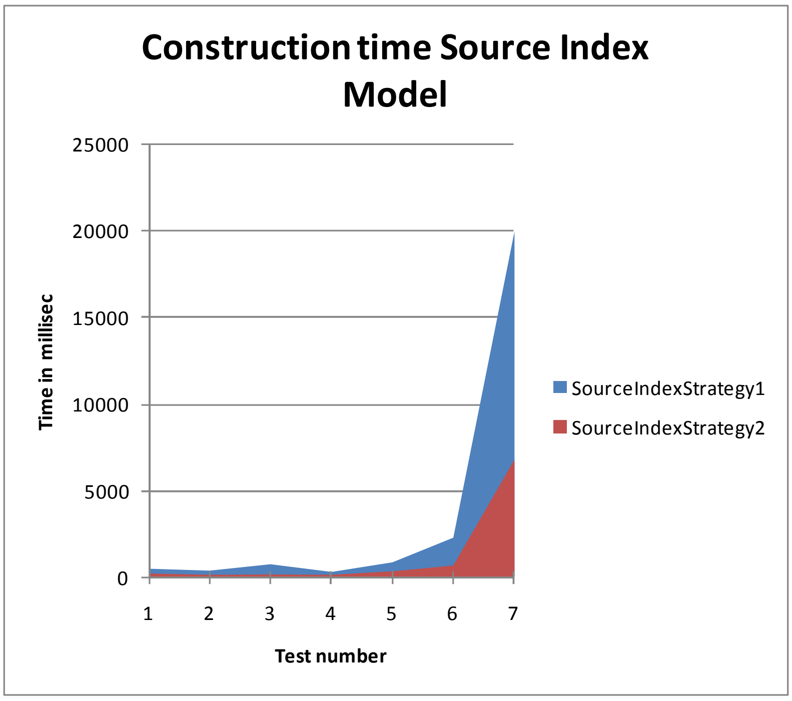


Chart 1: Construction time Source Index Model

For all seven tests, SourceIndexStrategy2 constructs the Source Index Model faster than SourceIndexStrategy1, the difference is in most cases quite large, for test 7 SourceIndexStrategy1 needs two to three times more time than SourceIndexStrategy2. This was expected, since SourceIndexStrategy1 keeps more source index information about the encountered data sources than SourceIndexStrategy2; as a result, more information must be extracted from the sources and stored in the Model.

8.2.3 Criterion 2: Size

This second criterion measures the size of the Source Index Model for SourceIndexStrategy1 and SourceIndexStrategy2. The size is measured in bytes (the Source Index Model is stored in N-Triple format). Chart 2 shows the test result for the seven test cases in the form of a bar diagram. The horizontal axis shows the test number, while the vertical axis represents the size in bytes.

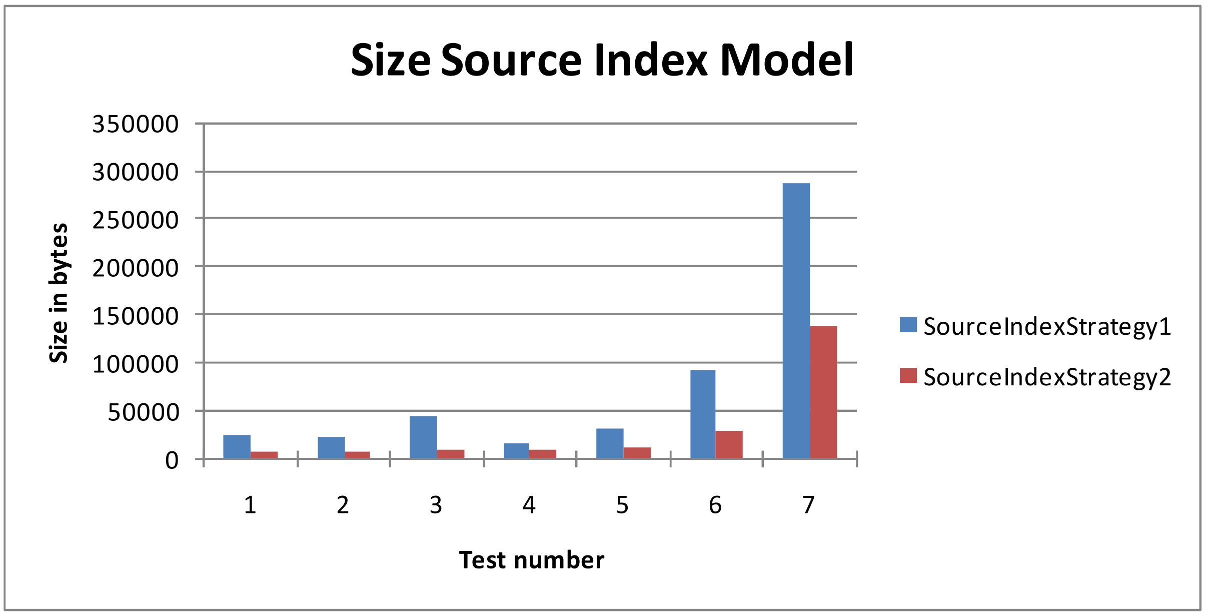


Chart 2: Size Source Index Model

When comparing SourceIndexStrategy1 and SourceIndexStrategy2, we can conclude that the Source Index Model of SourceIndexStrategy1 is always larger than the Source Index Model of SourceIndexStrategy2. This was expected since SourceIndexStrategy1 maintains more source index information; it also includes the used domains, than SourceIndexStrategy2. The difference is actually very big, for more explanation why the difference is so large see Chapter 5.

The Source Index Model of test 6 contains the source index information on four encountered data sources, while the Source Index Models of test 1 to 4 respectively contain source index information on one of the four encountered sources from test 6. Table 1 shows us that the Source Index Model for the four encountered data sources in test 6 is smaller than the Source Index Models of the individual encountered data sources (test 1 to 4) combined. This is expected, as both of the source index strategies try to reuse as much source index information as possible (sections 5.3.1 and 5.3.2). For instance, consider two files using the property "`foaf:name`" and a Source Index Model, formed by SourceIndexStrategy2, which already contains the source index information on one of these files. When adding the summary information of the second file to this Source Index Model, summary information stating that "`foaf:name`" is a property will not be added again (of course, the fact that the second file uses this property will still be added). The same observation can be made for test 5.

We can therefore conclude that an increase in similarity between the data sources leads to an increase in reuse of source index information, and therefore to a smaller Source Index Model. This conclusion is valid for SourceIndexStrategy1 as well as for SourceIndexStrategy2.

nr	SourceIndexStrategy1	SourceIndexStrategy2
1	24894	7198
2	23398	6764
3	43492	9292
4	15878	9942
5	32478	11052
6	91556	29994
7	286206	139118

Table 1: Size Source Index Models

8.2.4 Criterion 3: Time needed to find the relevant sources

This section evaluates the source index strategies to determine the amount of time necessary to find the relevant sources for a query. For each of the seven test cases, a Source Index Model was constructed, and the three corresponding aforementioned queries (see sections 8.2.1.1) were analyzed to determine the relevant sources. Chart 3 shows the results for the seven test cases: for each test case, the relevant sources for a low (test a), medium (test b) and high (test c) complexity query were retrieved. The horizontal axis shows the test number (together with the query number), while the vertical axis represents the time (in ms) necessary to find the relevant sources. Chart 4 shows the same results but only for the tests 4 to 6.

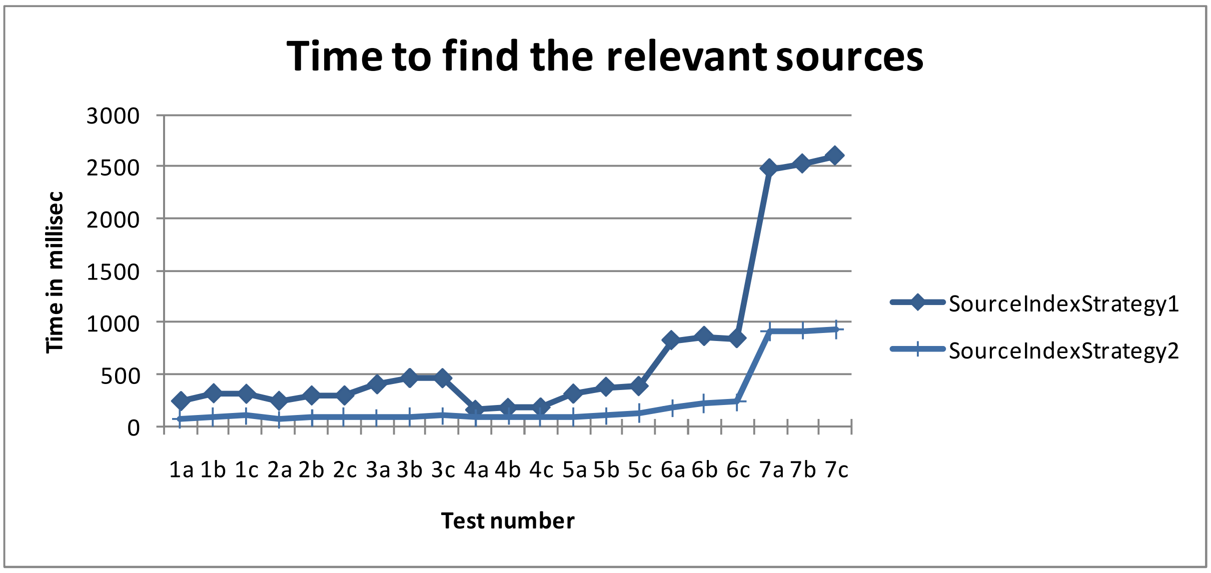


Chart 3: Time needed to find relevant sources

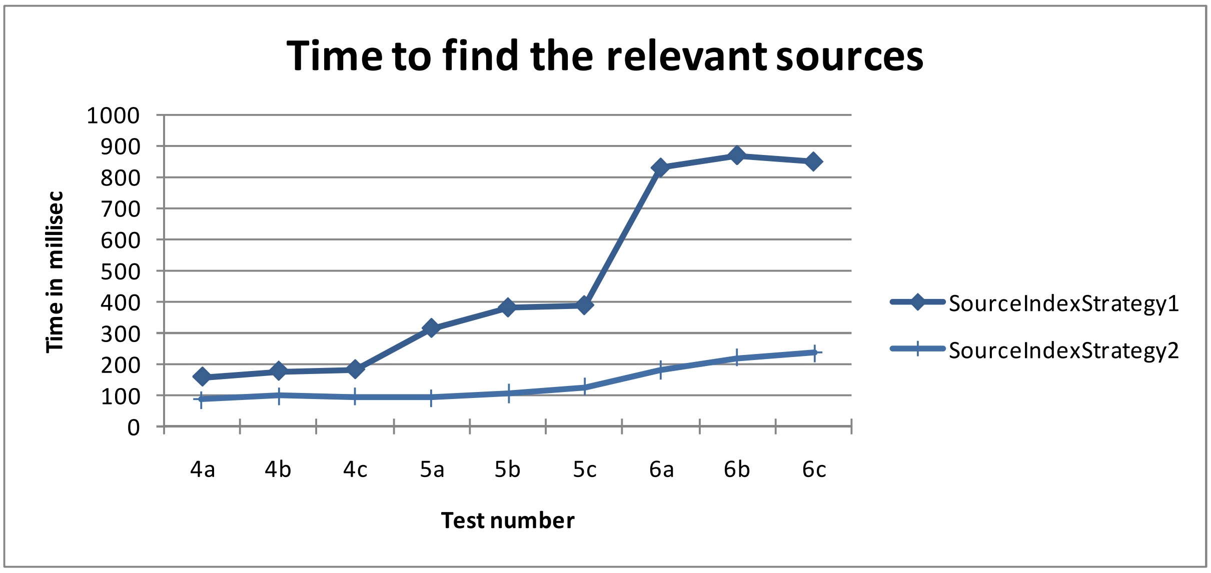


Chart 4: Time needed to find relevant sources (tests 4-6)

A first conclusion that can be drawn for the charts is related to the complexity of the three given queries per test case. More specifically, the time necessary to find the relevant sources rises with the complexity of the given query; the more complex the given query is, the more time the strategy needs to determine the relevant sources. The underlying reason is that more complex queries usually employ more predicates and domains; therefore, the Source Query (i.e., the query posed to the Source Index Model to determine the relevant sources, see section 5.4) contains more triple patterns to match these predicates and domains to encountered sources, and thus takes longer to execute. This conclusion is valid for both SourceIndexStrategy1 and SourceIndexStrategy2.

A second conclusion concerns the amount of summary information in the Source Index Model. We have learned from the previous section that keeping more summary information leads to a larger Source Index Model. Clearly, a larger Source Index Model will take longer to query, and therefore lead to an increased amount of time necessary to determine the relevant sources (this is especially apparent as the number of sources increase).

Comparing SourceIndexStrategy1 and SourceIndexStrategy2, the chart shows us that the time necessary to find the relevant sources is higher when using SourceIndexStrategy1 than using SourceIndexStrategy2. A first reason is that the Source Index Model of SourceIndexStrategy1 is larger than the Source Index Model of SourceIndexStrategy2, and therefore takes longer to query. A second reason is that the Source Index Query posed to the Source Index Model of SourceIndexStrategy1 will contain more triple patterns, since it also considers the domains of the predicates, while the Source Index Query posed to SourceIndexStrategy2 only considers the predicates themselves (see section 5.4 for more information). As a result, this more complex query will take longer to execute.

8.2.5 Criterion 4: Number of relevant sources

This section compares the number of relevant sources selected by the different Source index strategies. As in the previous section, for each test case the Source Index Model is constructed and the relevant sources are determined for the three aforementioned queries with a low (test a), medium (test b) and high (test c) complexity. To serve as a best- and worst-case scenario, two additional (hypothetical) strategies are taken into account: SourceIndexStrategyNone, where all the sources are considered relevant, and SourceIndexStrategyBest, where only the absolutely relevant sources are considered[30].

Chart 5 shows the results of all four Source index strategies in a bar chart, Chart 6 only shows the results of SourceIndexStrategy1, SourceIndexStrategy2 and SourceIndexStrategyBest. The horizontal axis shows the test number and the vertical axis the number of relevant sources. From these diagrams, we can conclude that SourceIndexStrategy1 and SourceIndexStrategy2 (and obviously SourceIndexStrategyBest) can prune out a lot of irrelevant sources. This is particularly the case for test case 7.

[30] These relevant sources were determined manually.

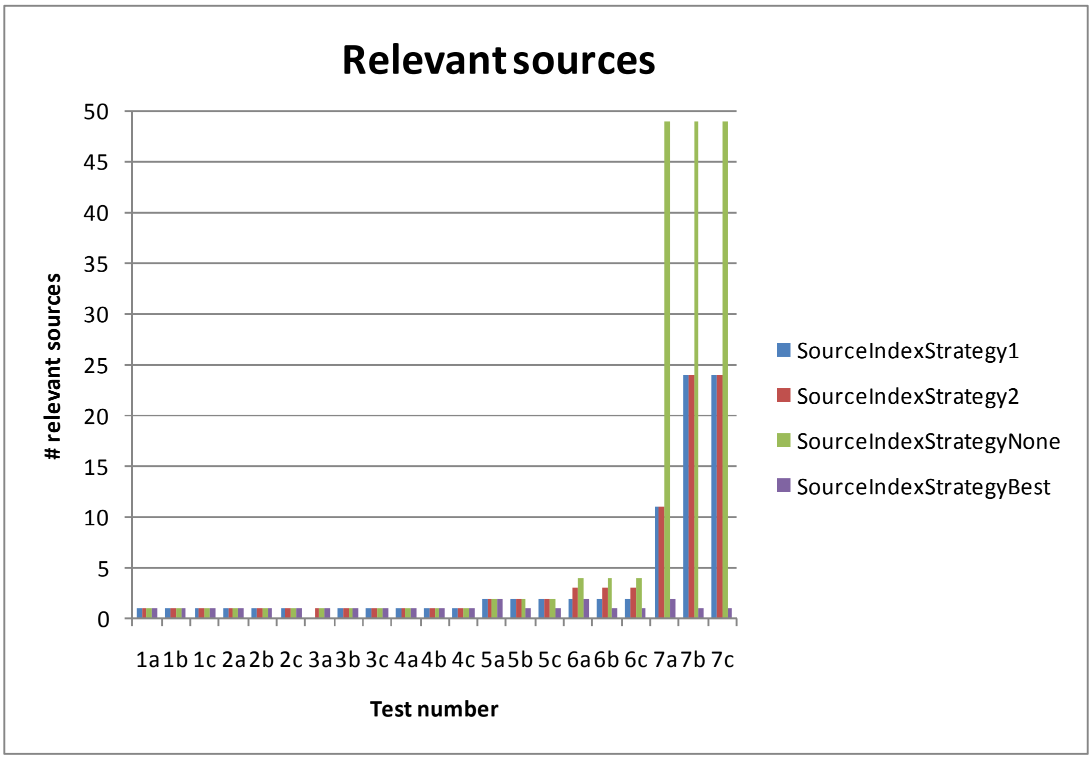


Chart 5: Number of relevant sources (1)

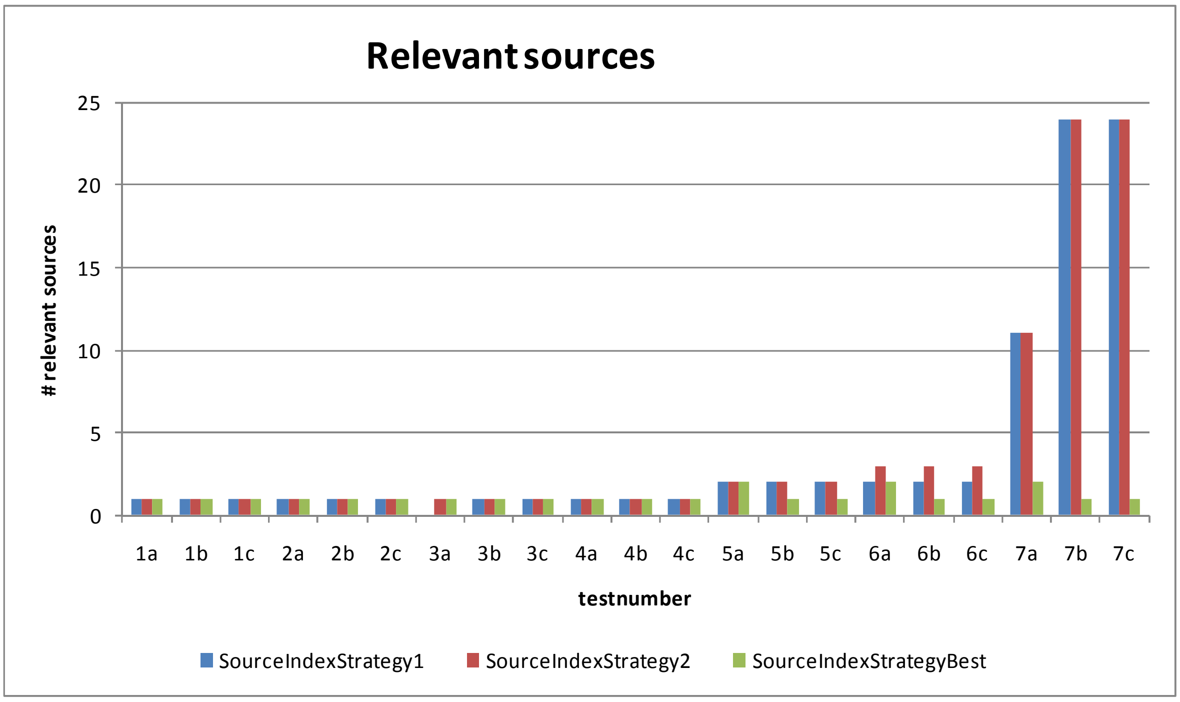


Chart 6: Relevant sources (2)

One could expect that when the complexity of a given query rises, the number of relevant sources will decrease since more constraints should be taken into consideration. However in our particular scenario (using SourceIndexStrategy1 and SourceIndexStrategy2) the number of relevant sources stays the same or increase when the complexity of the query rises. As explained in section 5.4 our source index strategies should return all sources which might solve the query or *a part* of the query, since combining (partial) relevant source may result in (new) actual query results. So the more the complexity of the query rises, the more domains and / or predicates that will be used, the more relevant sources that may be discovered. Test 7 shows that the number of relevant sources increases along with the complexity.

The charts show that Tests 1-4 do not favor one particular strategy: every query that is posed selects exactly the one source which is summarized by the Source Index Model. In test 5b and 5c, only 1 source is actually necessary to solve the posed query (as “detected” by SourceIndexStrategyBest), while SourceIndexStrategy1 and SourceIndexStrategy2 return two relevant sources. The second source, which is marked irrelevant by SourceIndexStrategyBest but not by SourceIndexStrategy1 and SourceIndexStrategy2, does not contain any results for the query, however this second source uses some of the predicates and / or domains mentioned in the query. SourceIndexStrategy1 and SourceIndexStrategy2 do not know with their little summary information that these triples used in the source do not solve the query, i.e., the values of those triples do not match the query.

The chart shows that SourceIndexStrategy1 and SourceIndexStrategy2 almost always return the same amount of relevant sources except for test 6a, 6b and 6c. In these particular cases SourceIndexStrategy1 uses the extra domain information to exclude an extra source. For example take the low complexity query for test 6, this query selects the title of the music records. SourceIndexStrategy1 uses the domain and predicate information to find all the sources that use the predicate “`dc:title`” with as domain “`mo:record`”. There are only two sources that match. While SourceIndexStrategy2 cannot use the domain information, it will find three sources relevant for the query, it cannot exclude the third source which uses the predicate “`dc:title`” but not with the domain “`mo:Record`”. So SourceIndexStrategy1 is more selective than SourceIndexStrategy2.

Now let’s consider one extra test to show the selectivity benefit of SourceIndexStrategy1. We have executed the low complexity query of test 6, this time on all the fifty encountered sources. SourceIndexStrategy2 will be able to reduce the fifty sources to only 9 relevant sources SourceIndexStrategy 1 however reduces to only 2 relevant sources, SourceIndexStrategyBest also marked 2 sources as being relevant. This is a major benefit for SourceIndexStrategy1.

However, SourceIndexStrategy1 can also wrongfully exclude sources that can still yield query results when they are combined (so-called false-negatives). These false-negatives occur when one source indicates the type of a certain resource, while another source uses the same resource as subject of a predicate used in the query, both sources are falsely considered irrelevant. In that case, SourceIndexStrategy1 will not detect the latter source as relevant, as it expects the type of a subject resource (i.e., “actual” domains of predicates, see section 5.4) to be specified in the same source as they occur as subject of a predicate. However, we expect that the amount of false-negatives is rather small, since the typing of a subject resource and their use with predicates usually occur in the same source. Future work (see Chapter 9) will be investigating how often these false-negatives occur in practice and how they can be avoided.

8.2.6 Criterion 5: Execution time of the original query

This section compares the execution time of the original (given) query on the RDF Graph consisting of the relevant sources. Before this test, the Source Index Model has been constructed, the relevant sources for the given query have been determined, and the relevant sources have been combined into one RDF Graph. The original query is executed for all seven test cases, each time with a low (test a), medium (test b) and high (test c) complexity query. The test cases are executed for SourceIndexStrategy1, SourceIndexStrategy2, SourceIndexStrategyNone (the worst case scenario) and SourceIndexStrategyBest (the best case scenario). Chart 7 shows the results for the four source index strategies with on the horizontal axis the execution time in milliseconds and on the vertical axis the test number.

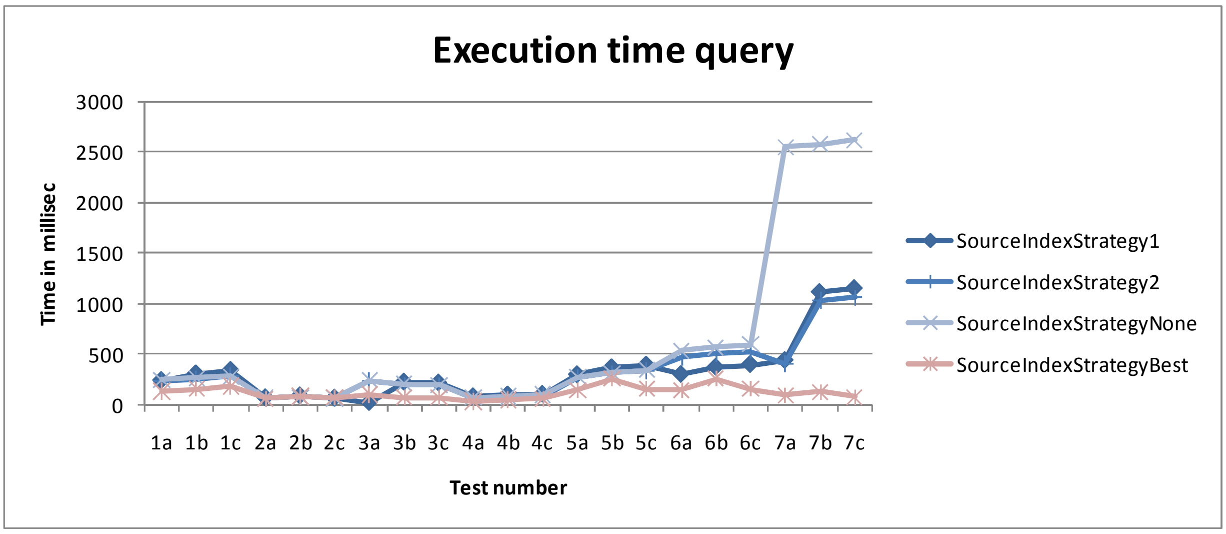


Chart 7: Execution time of the original query

From the previous section we know that SourceIndexStrategyNone selects more relevant sources, thus we expect that execution time of the query will be higher than for the other Source index strategies. This is proven by Chart 7.

Since SourceIndexStrategyBest selects less relevant sources, it is logical that the execution time of the query will be lower than those of the other source index strategies as Chart 7 shows. It can be seen on the graph that, however the execution time for SourceIndexStrategy1 and SourceIndexStrategy2 is lower than the worst case scenario, there is still room for improvement. It can be observed that the execution time for SourceIndexStrategy1 and SourceIndexStrategy2 is (more or less) the same for most of the test cases, as their selectivity (i.e., number of relevant sources returned) is also equivalent in those test. There is however one test where the execution time is lower for SourceIndexStrategy1 than for SourceIndexStrategy2, this is test 6, as the selectivity in that test was also lower for SourceIndexStrategy1 than for SourceIndexStrategy2.

A final conclusion from Chart 7 is that when the complexity of a query rises, the execution of the query rises as well. Reason is that the more complex the query is, the more constraints need to be considered, the more complicated the query resolving process is, the more time it takes.

8.2.7 Conclusion

From the first two sections, we have learned that SourceIndexStrategy2 performs better than SourceIndexStrategy1 regarding Source Index Model construction time and size.

Chart 8 shows the total execution time (i.e., the time necessary to determine the relevant sources together with the execution time of the original query) measured for SourceIndexStrategy1, SourceIndexStrategy2 and SourceIndexStrategyNone. This chart does not mention SourceIndexStrategyBest since the determination of the relevant sources is done manually and thus the determination time is not known. Tests 1-6 indicate that using a Source Index Model leads to lower performance than when no Source Index Model is used (SourceIndexStrategyNone). However, the amount of sources summarized in the Source Index Model for those tests is very small; it is expected that the user will encounter many more sources, as is the case in test 7. The Test 1-6 were thus employed to make a full comparison of the strategies, but do not represent realistic use cases.

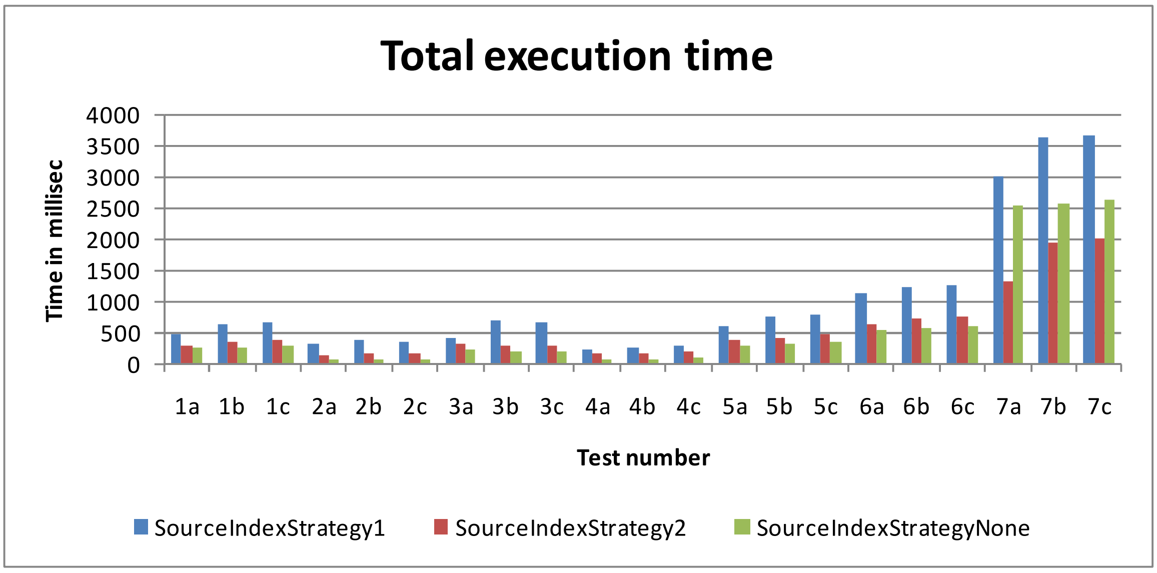


Chart 8: Total execution time

From the more realistic test 7, we can conclude that SourceIndexStrategy2 represents a significant improvement over SourceIndexStrategyNone, and performs much better than SourceIndexStrategy1 since the time to find the relevant data is much lower for SourceIndexStrategy2 than for SourceIndexStrategy1 and SourceIndexStrategy1 cannot be more selective (except for test 6) then SourceIndexStrategy2. In the section discussing the relevant sources criterion an extra test is mentioned, one in which SourceIndexStrategy1 only selects two sources as relevant while SourceIndexStrategy2 selects nine sources as relevant so the difference is quite big. The corresponding find time and query time for SourceIndexStrategy1 is 2618 and 315 respectively, while for SourceIndexStrategy2 the find time and query time are 1117 and 919. So even in this case SourceIndexStrategy1 is much more selective, the advantage is vanished by the high find time.

We do also need to consider the amount of time that is necessary to construct the Source Index Model in the case of SourceIndexStrategy1 and SourceIndexStrategy2 but the more times the Source Index Model is used, the more this time is negligible.

The final conclusion is that SourceIndexStrategy2 is the best strategy, it is not only the quickest strategy, it also takes the least space. SourceIndexStrategy1 takes more space and is a slower strategy despite the fact that it is more selective in indicating the relevant sources.

8.3 Cache Strategy

The previous section evaluated the developed source index strategies, which maintain source index information on the encountered data sources. This section evaluates the second part of this thesis, where two caching strategies were developed. The evaluation takes into account the following criteria: time necessary to collect the relevant data; time necessary to execute the query on the relevant data; time necessary to download the relevant sources; time necessary to update the cache when a new source is encountered and time necessary to update the cache (when the sources are accessed). The first section provides details on the employed test scenario, followed by sections evaluating the strategies with regards to the criteria above. The last section provides conclusions that can be drawn from the evaluation of the strategies.

In our first caching mechanism, CacheStrategySource, entire data sources are cached, while in CacheStrategyPredicate only triples using a particular predicate are cached. In addition, we consider a worst-case scenario called CacheStrategyNone (i.e., none of the encountered sources are cached and thus need to be downloaded every time they are requested), and a best-case scenario where all encountered data sources are cached called CacheStrategyBest. The latter two are thus two hypothetical caching mechanisms, employed to compare the two developed mechanisms to a worst- and best-case scenarios.

8.3.1 Test scenario

One test scenario is used in order to evaluate the developed caching mechanisms. This scenario consists of three parts (which are all described in 8.3.1.1), while every part consists of the same seven queries of which the complexity varies from low to high.

The test scenario is executed for each cache strategy (i.e., CacheStrategyPredicate, CacheStrategySource, CacheStrategyNone and CacheStrategyBest) and criteria. For CacheStrategyPredicate and CacheStrategySource, the scenario is executed three times, each time with a different cache size. For more information about the actual cache size and how it is determined see 8.3.1.2.

Each cache strategy use sSourceIndexStrategy2 as Source Index Strategy to determine the relevant sources, since we concluded from the evaluation section of source index strategies that it is the better one of the two.

8.3.1.1 Test scenario composition

The test scenario consists of three parts, which are executed immediately one after another. The first part has as purpose to build up the cache, and executes seven queries. It is possible that a query can benefit from the cache build up by one of the previously executed queries (e.g., Query 4 might benefit from Query 3). During this test, new data sources are encountered (in total fifty sources). This allows us to measure how good the caching mechanism is while building up the cache and still encountering new sources. The order of encountering and executing is as followed:

- encounter one new source,
- execute Query 1,
- encounter two new sources,
- execute Query 2,
- encounter eight new sources,
- execute Query 3,
- encounter four new sources,
- execute Query 4,
- encounter seven new sources,
- execute Query 5,
- execute Query 6,
- execute Query 7,
- encounter twenty-eight new sources.

The second part executes the same seven queries in the same order as in the first part; however, this time the cache is already built (via the queries from the first part), and no new data sources are encountered. This part allows us to measure how good the cache mechanism is with a built cache. However, not all relevant information will already be in the cache, some queries for example for Query 3 will in the second part have much more relevant sources than in the first because we have encountered much more sources. Therefore some sources might be available in the cache and others won't.

The third and last part again executes the seven queries in the same order as before, while the cache built in parts I and II is retained and no new sources are encountered. This part allows us to compare the results of the second part without other issues interfering (i.e., no newly encountered sources and no new relevant sources for a particular query).

8.3.1.2 Determine the cache sizes

In this thesis two caching mechanism were developed: CacheStrategySource, which caches entire encountered sources, and CacheStrategyPredicate, which caches triples of the encountered sources using a particular predicate. The chosen test scenario is be executed on each of them three times, each time with a different cache size.

The tests were once executed without cache size limitation to determine the 25th percentile, 50th percentile, and 75th percentile of the amount of bytes that are relevant for the queries, these percentiles represent the three used cache size. These sizes are not the same for CacheStrategyPredicate and CacheStrategySource since CacheStrategyPredicate selects a much smaller set of triples to actually perform the query on than CacheStrategySource, as

CacheStrategyPredicate only selects triples using the relevant predicates, while CacheStrategySource selects all the triples of sources containing a relevant predicate. Table 2 shows the relevant bytes for each query and cache strategies.

For CacheStrategySource, the 25th percentile is 18753, the 50th percentile is 47836 and the 75th percentile is 90169. For CacheStrategyPredicate, the 25th percentile is 3205, the 50th percentile is 7082 and the 75th percentile is 10258. This means that the cache size for CacheStrategyPredicate is five to eight times smaller than for CacheStrategySource.

Part I	**CacheStrategySource relevant bytes**	**CacheStrategyPredicate relevant bytes**
Query 1	7.748	138
Query 2	10.164	2.394
Query 3	17.414	2.802
Query 4	31.592	6.070
Query 5	64.080	8.094
Query 6	74.888	9.970
Query 7	74.888	9.844
Part II & III		
Query 1	7.748	138
Query 2	22.934	4.414
Query 3	22.768	4.568
Query 4	95.262	10.354
Query 5	95.262	10.354
Query 6	112.012	12.660
Query 7	112.012	12.588

Table 2: Relevant amount of data

8.3.1.3 The seven queries

As explained previously, the test scenario consists of three parts where in each part the same seven queries are executed in the same order. This subsection explains the seven used queries (in the order in which they are executed). For each query, the meaning and complexity (low, medium or high) is mentioned. A low complexity query is a basic query that consists of a select and where clause with one or two basic triple patterns. A medium complexity query is composed out of a select and where clause with five to six where triple patterns. A union query with filters and a total of ten to twelve where triple patterns is considered to be a high complexity query.

This first query is a low complexity query which selects the nicknames.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE
{
  ?p foaf:nick ?name .
}
```

SPARQL query 22: Query 1

This second low complexity query shows all the topics in which people are interested.

```
PREFIX foaf: <http://xmlns.com/foaf/spec/>
SELECT ?topic
WHERE
{
   ?p foaf:interest ?interests .
   ?interests foaf:topic ?topic .
}
```

SPARQL query 23: Query 2

The third query shows all the given names and surnames. This query has again a low complexity.

```
PREFIX foaf: <http://xmlns.com/foaf/spec/>
SELECT ?givenname ?surname
WHERE
{
   ?p foaf:givenname ?givenname .
   ?p foaf:surname ?surname .
}
```

SPARQL query 24: Query 3

The fourth query is a medium complexity query which shows the given name, surname, birthday, biography and a photo of all people.

```
PREFIX foaf: <http://xmlns.com/foaf/spec/>
PREFIX bio: <http://purl.org/vocab/bio/0.1/>
PREFIX vcard: <http://www.w3.org/2006/vcard/ns#>
SELECT ?givenname ?surname ?bday ?bio ?img
WHERE
{
   ?p foaf:givenname ?givenname .
   ?p foaf:surname ?surname .
   ?p vcard:bday ?bday .
   ?p bio:olb ?bio .
   ?p foaf:img ?img .
}
```

SPARQL query 25: Query 4

This fifth query shows again the given name, surname, birthday, biography and photo, but only for people of whom the surname starts with ‘a’ or ‘b’ (case insensitive). This query has a high complexity.

```
PREFIX foaf: <http://xmlns.com/foaf/spec/>
PREFIX bio: <http://purl.org/vocab/bio/0.1/>
PREFIX vcard: <http://www.w3.org/2006/vcard/ns#>
SELECT ?givenname ?surname ?bday ?bio ?img
WHERE
{
   {
      ?p foaf:givenname ?givenname .
      ?p foaf:surname ?surname .
      ?p vcard:bday ?bday .
      ?p bio:olb ?bio .
      ?p foaf:img ?img .
      FILTER(regex(str(?surname) ,'^a',  'i'))
   }
   UNION
   {
      ?p foaf:givenname ?givenname .
      ?p foaf:surname ?surname .
      ?p vcard:bday ?bday .
      ?p bio:olb ?bio .
      ?p foaf:img ?img .
      FILTER(regex(str(?surname) ,'^b',  'i'))
   }
}
```

SPARQL query 26: Query 5

The sixth query is a medium complexity query, which shows the given name, surname, city and country of birth for all people.

```
PREFIX foaf: <http://xmlns.com/foaf/spec/>
PREFIX bio: <http://purl.org/vocab/bio/0.1/>
PREFIX it: <http://daml.umbc.edu/ontologies/ittalks/address#>
SELECT ?givenname ?surname ?city ?country
WHERE
{
   ?p foaf:givenname ?givenname .
   ?p foaf:surname ?surname .
   ?p bio:event ?birth .
   ?birth a bio:Birth .
   ?birth bio:place ?place .
   ?place it:city ?city .
   ?place it:country ?country .
}
```

SPARQL query 27: Query 6

This seventh and last query has a high complexity. It selects the given name, surname, city and country of birth for all people that are born in the USA or Austria.

```
PREFIX foaf: <http://xmlns.com/foaf/spec/>
PREFIX bio: <http://purl.org/vocab/bio/0.1/>
PREFIX it: <http://daml.umbc.edu/ontologies/ittalks/address#>
SELECT ?givenname ?surname ?city ?country
WHERE
{
   {
      ?p foaf:givenname ?givenname .
      ?p foaf:surname ?surname .
      ?p bio:event ?birth .
      ?birth a bio:Birth .
      ?birth bio:place ?place .
      ?place it:city ?city .
      ?place it:country ?country .
      FILTER(str(?country)='USA')
   }
   UNION
   {
      ?p foaf:givenname ?givenname .
      ?p foaf:surname ?surname .
      ?p bio:event ?birth .
      ?birth a bio:Birth .
      ?birth bio:place ?place .
      ?place it:city ?city .
      ?place it:country ?country .
      FILTER(str(?country)='Austria')
   }
}
```

SPARQL query 28: Query 7

8.3.2 Criterion 1: Time necessary to collect the relevant data

This first criterion is used to evaluate the developed caching mechanisms by measuring the time that is necessary to collect the relevant data for a given query. This amount of time includes the download time, time necessary to update the cache, the time necessary to retrieve some data out of the cache and time to construct the RDF Graph containing all the relevant data. This criterion mentions CacheStrategySource, CacheStrategyPredicate, CacheStrategyNone and CacheStrategyAll. The following subsections go into more detail on each of these included times; we discuss the evaluation of the total times here.

The chart below shows the amount of time necessary to get the relevant data for all the cache strategies. The horizontal axis shows the part and query number while the vertical axis shows the time necessary to collect the relevant data in milliseconds (CacheStrategy is abbreviated to “CS”). Note that the bar of CacheStrategyAll is difficult to see in most cases, as it often only needs a few milliseconds to get the sources’ data out of its cache.

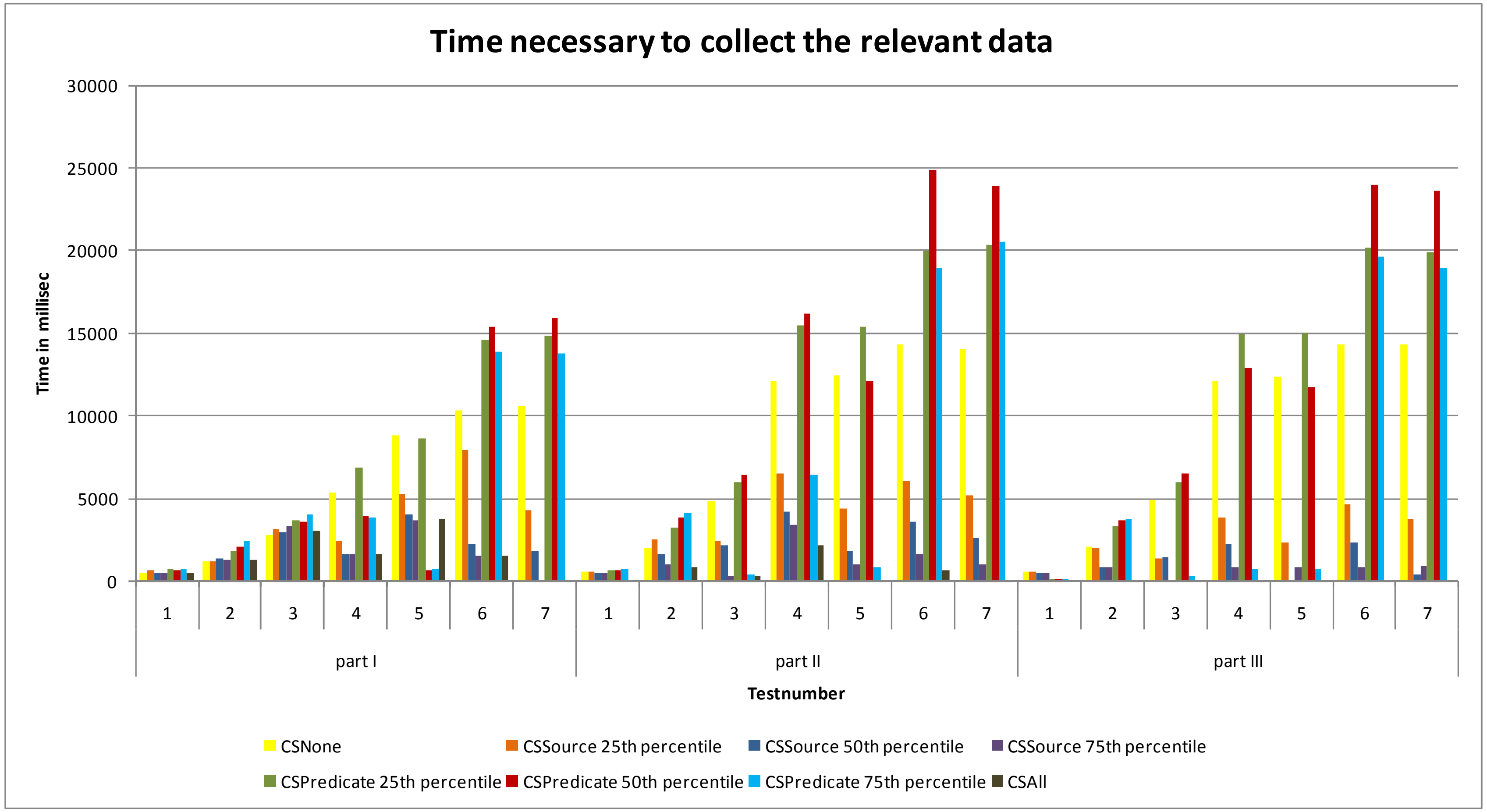


Chart 9: Time necessary to collect the relevant data

Chart 9 indicates that the time necessary to collect the relevant data is the highest for CacheStrategyPredicate; it is even higher than the time necessary for CacheStrategyNone, the worst case scenario in which all the relevant data must be downloaded every time it is requested. A first explanation for this result is that a lot of overhead is required when using CacheStrategyPredicate, e.g., the encountered data sources are translated into a special list structure, when the relevant data for a query is requested the list structure must be translated again to an RDF Graph, when new relevant sources are encountered the cached blank triples of that source must be adjusted … A second explanation is that the system must download the sources many times for the different predicates. See section 6.1.2 for more details on this issue.

From the previous paragraphs and charts, we learned that CacheStrategyPredicate does not perform well. Although all three cache sizes perform badly, it is still worth comparing them one to another. Chart 10 shows the necessary times for CacheStrategyPredicate with all three sizes. From the chart we can conclude that the 50th percentile cache strategy needs more time to get the relevant data than the 25th percentile except for part I Query 3, 4, 5; part II Query 5 and part III Query 4, 5. A second conclusion is that the 75th percentile performs the best, although in some cases the 25th percentile comes close to the performance of the 75th percentile (e.g., part I – query 6 and 7, …). If we consider speed versus stored data, the 25th percentile is the better option.

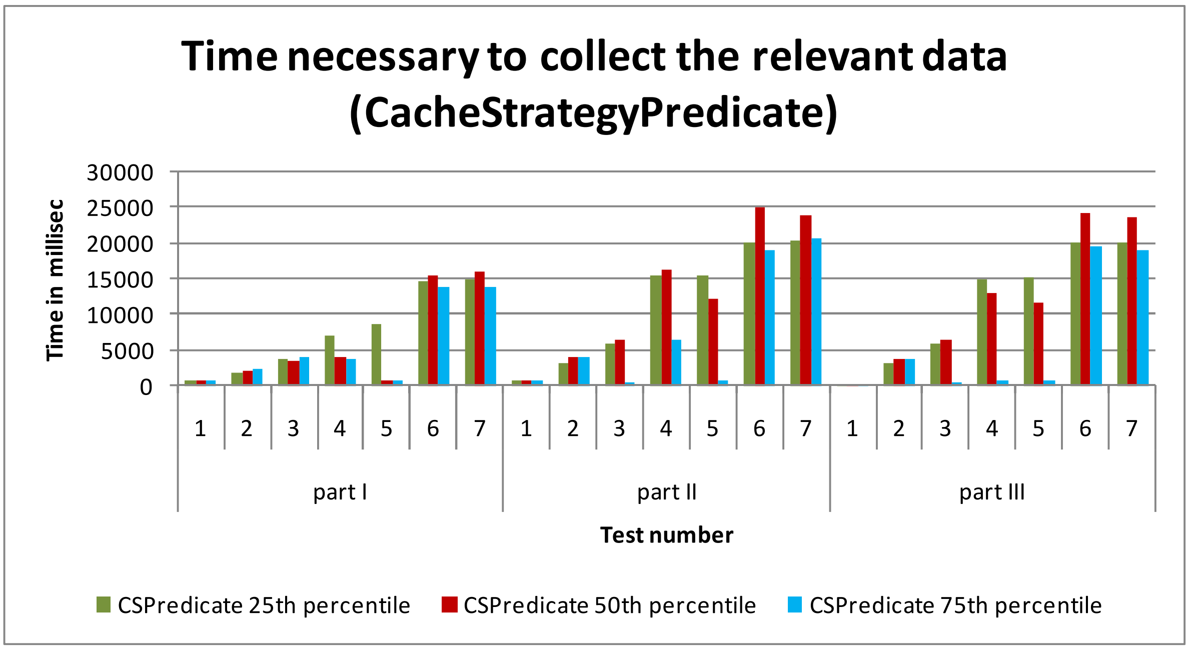


Chart 10: Time necessary to collect the relevant data (CacheStrategyPredicate)

Contrary to CacheStrategyPredicate, CacheStrategySource performs quite well. The necessary time to get the relevant data is in most cases much lower than for CacheStrategyNone. Contrary to CacheStrategyPredicate, this cache strategy does not require much overhead, the sources do not need to be translated, the blank nodes of relevant sources do not need to be updated; furthermore, the system must download the sources only once for the different predicates (see ..)

The chart below shows the retrieval times for CacheStrategySource with the three cache sizes. The 75th percentile performs the best, which is more or less expected since it can store the most relevant sources; however, in some cases the difference with the 50th percentile is rather small (e.g. part I – Query 1-6; part II – Query 1, 2, 4, 5), in one case it was even quicker than the 75th percentile (part III Query 5). In this particular case, the 50th percentile has cached as much relevant information as the 75th percentile however since its cache is smaller, the time necessary to get the relevant data out of the cache is also smaller. The 25th percentile performs the worst; the difference with the 50th percentile is in most cases quite large especially for part II and III. From this we can thus conclude that a larger cache size results in higher performance. However, the performance of the 50th percentile cache size is in many cases comparable to that of the 75th percentile; therefore, it can be considered as a good trade-off between performance and cache.

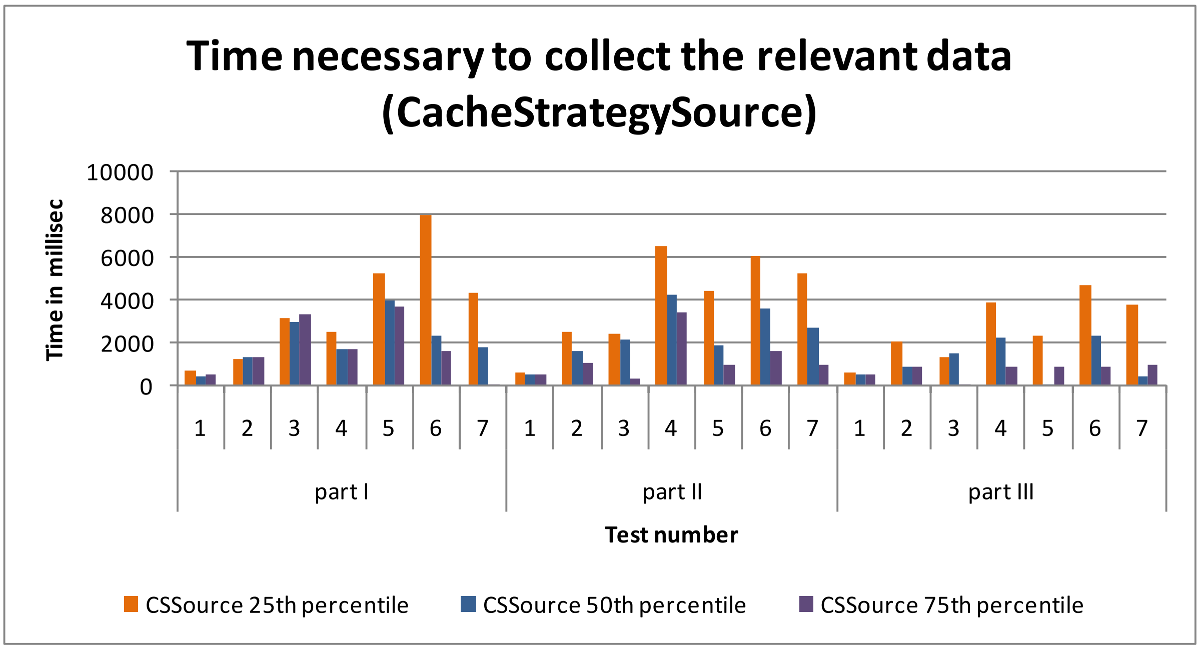


Chart 11: Time necessary to collect the relevant data (CacheStrategySource)

8.3.3 Criterion 2: Time necessary to download the uncached data

This third criterion, time necessary to download the uncached relevant data, is used to evaluate the cache strategies in terms of network performance. The chart shown on the next page shows the results for all cache strategies. The horizontal axis mentions the part and query number, while the vertical axis represents the time necessary to download the non-cached relevant data.

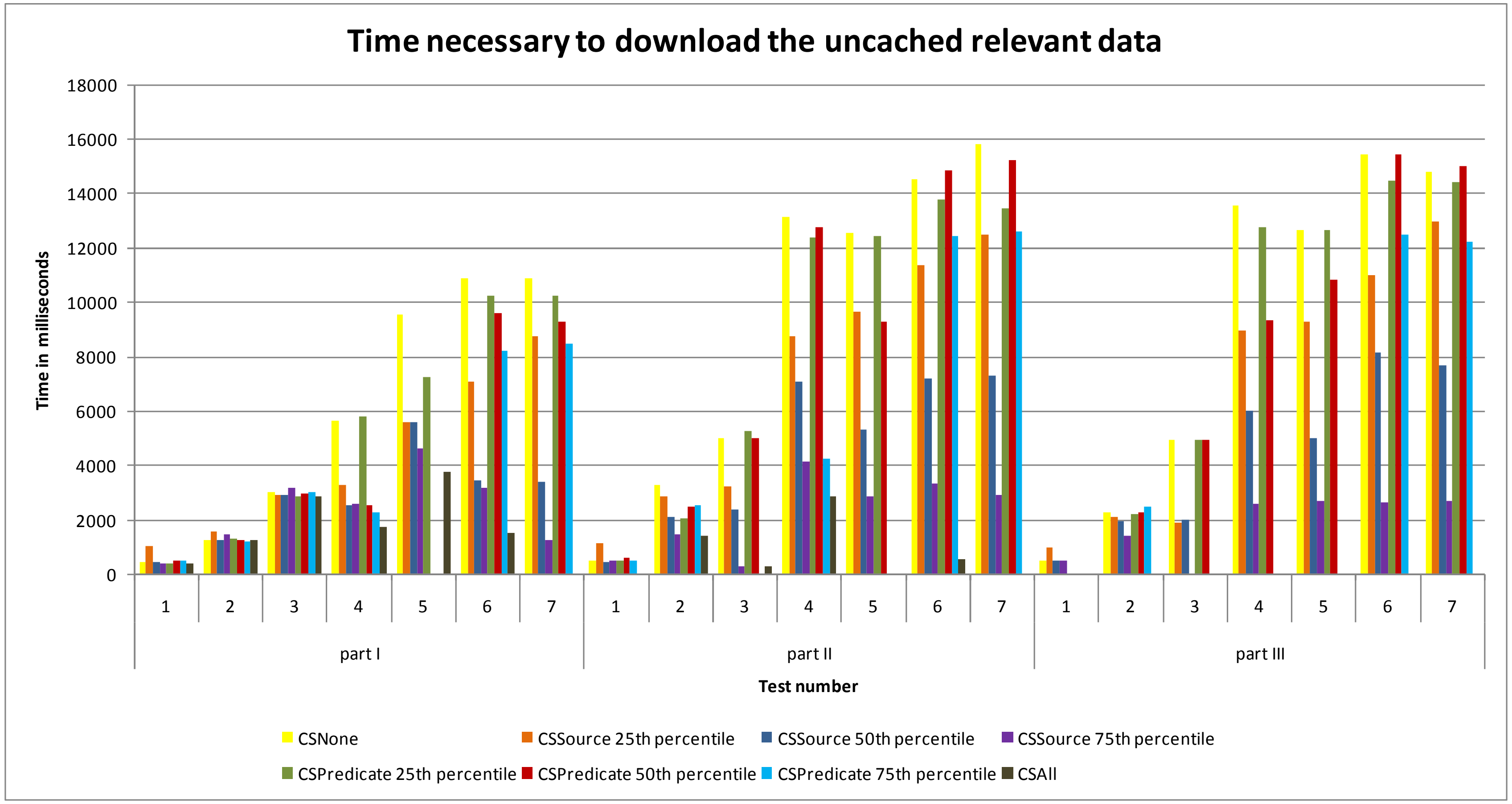


Chart 12: Time necessary to download the uncached data

From Chart 9 we concluded that CacheStrategyPredicate needs much more time to collect the relevant data compared to CacheStrategySource. One cause was that CacheStrategyPredicate creates a lot more overhead: sources need to be transformed into the list structure, blank node identifiers must be updated … (see section 6.1.2 for more information). A second cause is shown on Chart 12, which indicates that all three CacheStrategyPredicates need a lot more time to download the uncached relevant data; often, it even needs as much time is as SourceIndexStrategyNone, which means that all the relevant source are downloaded. Why does CacheStrategyPredicate need to download so much data? The executed queries often use predicates that appear in the same sources containing one of the previously used predicates. When such a (new) predicate is used, CacheStrategyPredicate needs to download the same sources again, in order to extract the triples containing the new predicate. On the other hand, CacheStrategySource caches data on a per-source level, and thus does not need to download the same sources again as they were already cached when the previous predicate was used.

For instance, consider Query 3, which selects the given names and surnames of the people, and Query 4 which selects the given names, surnames, birth days, biographies and images of the people. CacheStrategySource downloads all relevant sources for the first query, and caches all or some of them (depending on the size that is used). When CacheStrategySource is asked to collect the relevant data for Query 4, many of the cached sources of Query 3 will still be relevant for the predicates "`vcard:bday`", "`bio:olb`" and "`foaf:img`" used in Query 4. This means that for Query 4, not all sources relevant for these three new predicates need to be downloaded again; only those that were not cached before must be downloaded. Analogous to CacheStrategySource, CacheStrategyPredicate also downloads all relevant sources for "`foaf:givenname`" and "`foaf:surname`" in order to collect the relevant data for Query 3. However, in this case only the triples using one of those two predicates are actually cached. Therefore, when CacheStrategyPredicate collects the relevant data for Query 4, it needs to download all relevant sources for the predicates "`vcard:bday`", "`bio:olb`" and "`foaf:img`" again, also including the relevant sources which were previously downloaded for Query 3. As a result, much more data needs to be downloaded, and more time is needed to get the relevant data.

The SCOUT framework does not require encountered entities to have a query endpoint; they only need to make the information (e.g., by having some RDF data online) available. Suppose that the entities would have a query endpoint; in that case, CacheStrategyPredicate could be optimized to only ask for the RDF triples containing the predicates "`vcard:bday`", "`bio:olb`" and "`foaf:img`", which would lead to a lower amount of downloaded information. Moreover, we may expect that the download time will be much lower than for CacheStrategySource, and it might even be quicker in retrieving the relevant data altogether.

The chart below shows the time necessary to download the non-cached relevant data for CacheStrategySource with the three different cache sizes. The horizontal axis shows the part and query number while the vertical axis shows the amount of time that was needed.

What we see on this chart is that the 75th percentile needs a lot less time to download the relevant data. This was expected, since more sources can be cached (as it has a larger cache size); therefore, there is a higher chance to get a cache hit, and thus less data needs to be downloaded. In most cases, the difference with the 25th and 50th percentile is significant (especially compared to the 25th percentile). However, in some cases the difference with the 50th percentile is rather small (e.g., part I query 6, part II query 2).

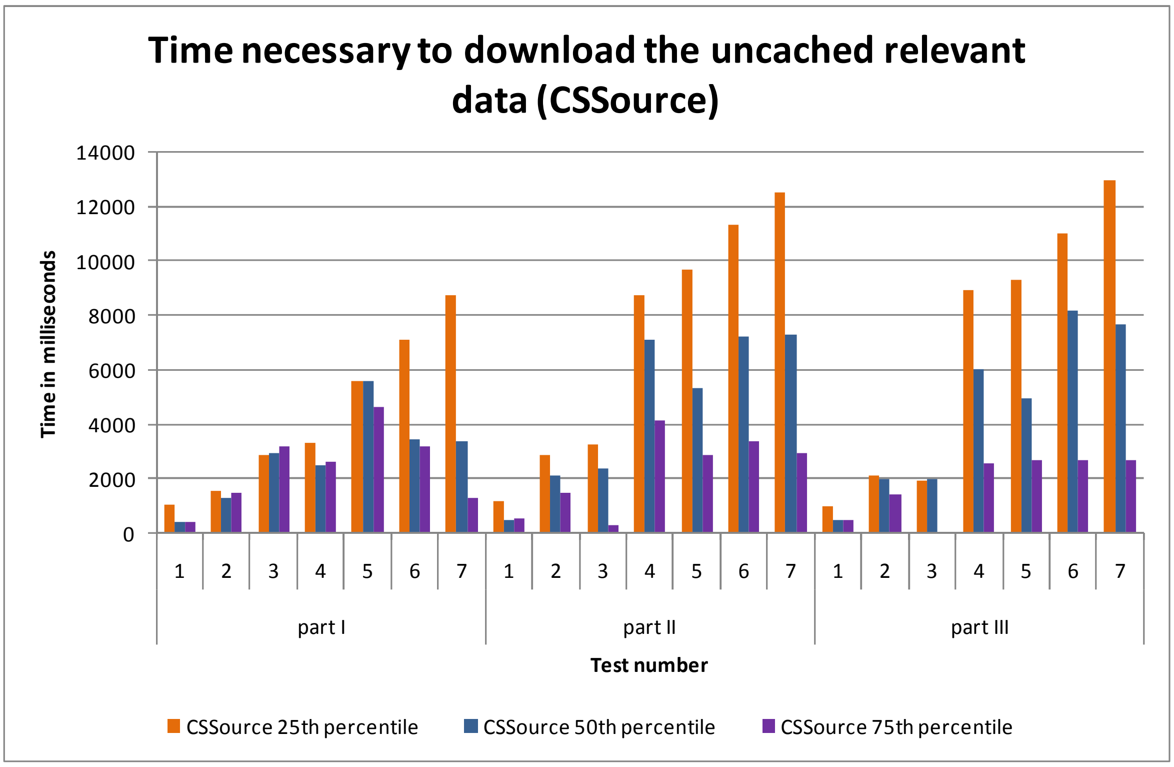


Chart 13: Time necessary to download the uncached data (CSSource)

8.3.4 Criterion 4: Time necessary to update the cache

This fourth criterion measures the amount of time that is spent on updating the cache. Chart 14 shows the measured times for CacheStrategySource and CacheStrategyPredicate for each of the three cache sizes. The horizontal axis mentions the part and query number while the vertical axis mentions the measured times in milliseconds.

The chart indicates that the update times of CacheStrategyPredicate are (relatively) much lower than for CacheStrategySource. Note that the measured times lie between zero milliseconds and twenty-seven milliseconds and that the difference is thus very small. A first reason is that a part of the created overhead of CacheStrategyPredicate is not included is the measured time, the time spent on constructing the relevant predicate graphs of the downloaded sources, since this is also part of the construction of the RDF Graph containing the relevant information for the query. See sections 6.1.2 and 7.6.2 for more information. A second reason is that the size of the CacheStrategyPredicate is much smaller, and thus less information needs to be maintained.

This chart shows that CacheStrategySource with as size the 50th percentile needs the most time to update the cache. The difference with the 25th can be explained by the fact that the 25th percentile has a much smaller cache size; as many relevant sources will not fit in this cache, it needs to consider fewer elements. The difference with the 75th percentile can be explained by the fact that the 75th percentile has more cache hits, in other words it retrieves more elements from the cache. And thus it won't need to add new elements to the cache or update the already cached elements as much as the other percentile.

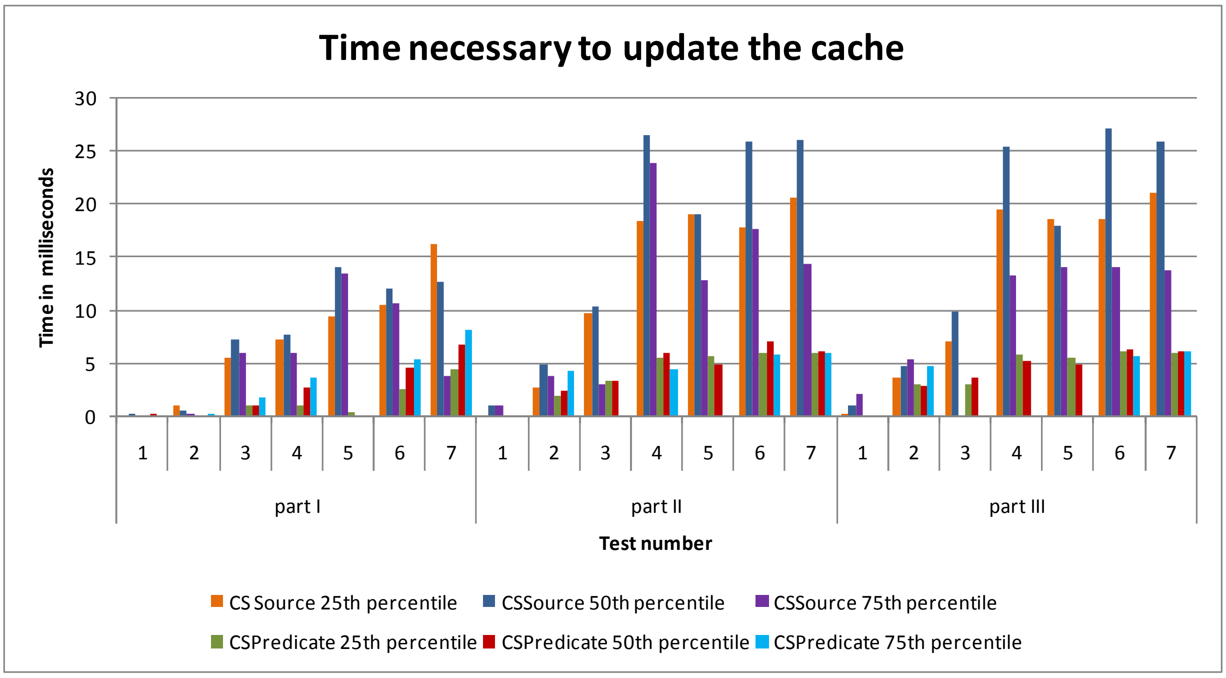


Chart 14: Time necessary to update the cache

8.3.5 Criterion 4: Time necessary when of a new source is discovered

This fourth and last criterion measures the amount of time that is necessary to update the cache when a new source is encountered. For CacheStrategySource this time is zero since nothing will be added to the cache, contrary to CacheStrategyPredicate where certain triples of the newly encountered source (which use a predicate present in the cache) need to be extracted and added to the appropriate cache entry.

The chart below shows the results for CacheStrategyPredicate for all three cache sizes. The horizontal axis shows the discovery number that occurs in the first part of the test scenario, while the vertical axis shows the measured time in milliseconds. Number 1 is the discovery of one source when no query has been executed yet; therefore, the cache is empty and nothing needs to be updated (this explains the fact that no time is necessary). Number 2 is the discovery of two new sources immediately after Query 1, number 3 encountered eight new sources after Query 2, number 4 is the discovery of four new sources after Query 3, number 5 is the encountering of seven new sources after Query 4 and finally number 6 is the encountering of twenty-eight new sources after Query 7.

The chart shows that the strategy needs the most time to update the cache when it has the 75th percentile as size. This was expected, since the 75th percentile size can cache the most predicates: as a result, more triples need to be added to the cache whenever a source is encountered. However, the difference with the other two percentiles is very small and not significant.

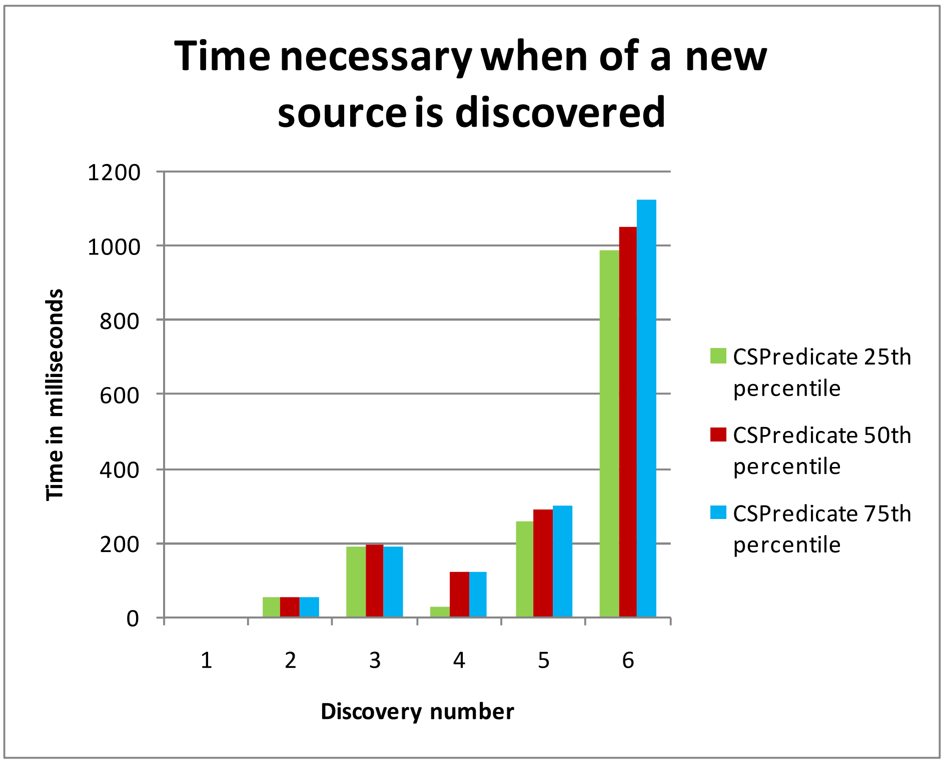


Chart 15: Time necessary when a new source is discovered

8.3.6 Criterion 5: Time necessary to execute the given query

This fifth and last evaluation criterion evaluates the caching mechanisms by measuring the amount of time that is necessary to execute the query on the relevant data. Since CacheStrategyPredicate only selects the triples that use a predicate from the query and not all the triples of a source using such a predicate, it will return a much smaller set (actually a subset) of relevant data (as illustrated by Table 2).

Chart 16 shows the amount of time necessary to execute the query. The horizontal axis shows the three parts of the test scenario with the corresponding query numbers while the vertical axis shows the amount of time necessary to execute the query on the relevant data (measured in milliseconds).

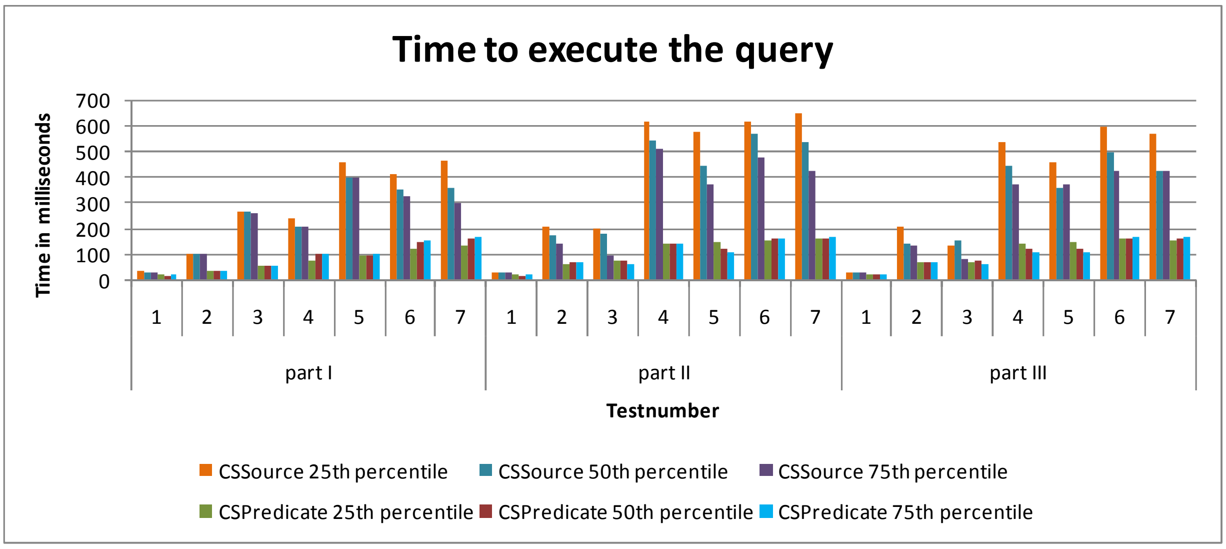


Chart 16: Time necessary to execute the given query

We thus expected that executing the query on relevant data returned by CacheStrategyPredicate would be much quicker than CacheStrategySource, since CacheStrategyPredicate returns a much smaller amount of relevant triples (see above). From the chart we see that the difference between CacheStrategyPredicate and CacheStrategySource is large (sometimes more than double), especially for the tests were the amount of relevant sources is quite high.

8.3.7 Conclusion

This section evaluated the cache strategies in terms of the time needed to collect the relevant data (which includes time needed to query the relevant data, and time needed to download the uncached relevant data), time needed to update the cache when the relevant data of a query must be retrieved and time needed to update the cache when a new source is encountered.

We can conclude that the performance of CacheStrategyPredicate is poor, mostly due to the amount of time it needs to download the relevant sources, the overhead caused by the composition of special list to store the RDF triples, the updating of the blank nodes … It cannot benefit from the fact that some sources have already been downloaded previously; i.e., any source containing triples for a new predicate needs to be downloaded again, even if that source had been previously downloaded in order to extract triples related to a different predicate. On the other hand, CacheStrategySource can retrieve previously downloaded and relevant sources straight from the cache. Another issue that should be taken into consideration is the amount of time necessary to update the cache with newly encountered sources. CacheStrategySource needs zero amount of time, since it does not update the cache with newly encountered sources, while CacheStrategyPredicate needs extra time to update the cache.

However, CacheStrategyPredicate returns a much smaller set of relevant data, which only contains the triples using a certain predicate used in the given query; in case of CacheStrategySource, the entire relevant sources are returned (see section 6.1.1). Additionally, CacheStrategyPredicate needs a lot less space than CacheStrategySource; more specifically, five to eight times as less (see section 6.1.2). These were actually the reasons this strategy was developed, as we expected they would lead to increased performance (see Chapter 6). Unfortunately, these issues do not outweigh the extra amount of time necessary to download relevant sources. Chart 17, which shows the sum of the time necessary to collect the relevant data and to query it, proves this; the total collection and execution times for the three CacheStrategyPredicates are much higher than those of CacheStrategySource. Finally, the free space gained by CacheStrategyPredicate certainly does not measure up to the loss in performance; especially considering the fact that in many of the cases (certainly with a high query complexity), CacheStrategyNone even outperforms CacheStrategyPredicate.

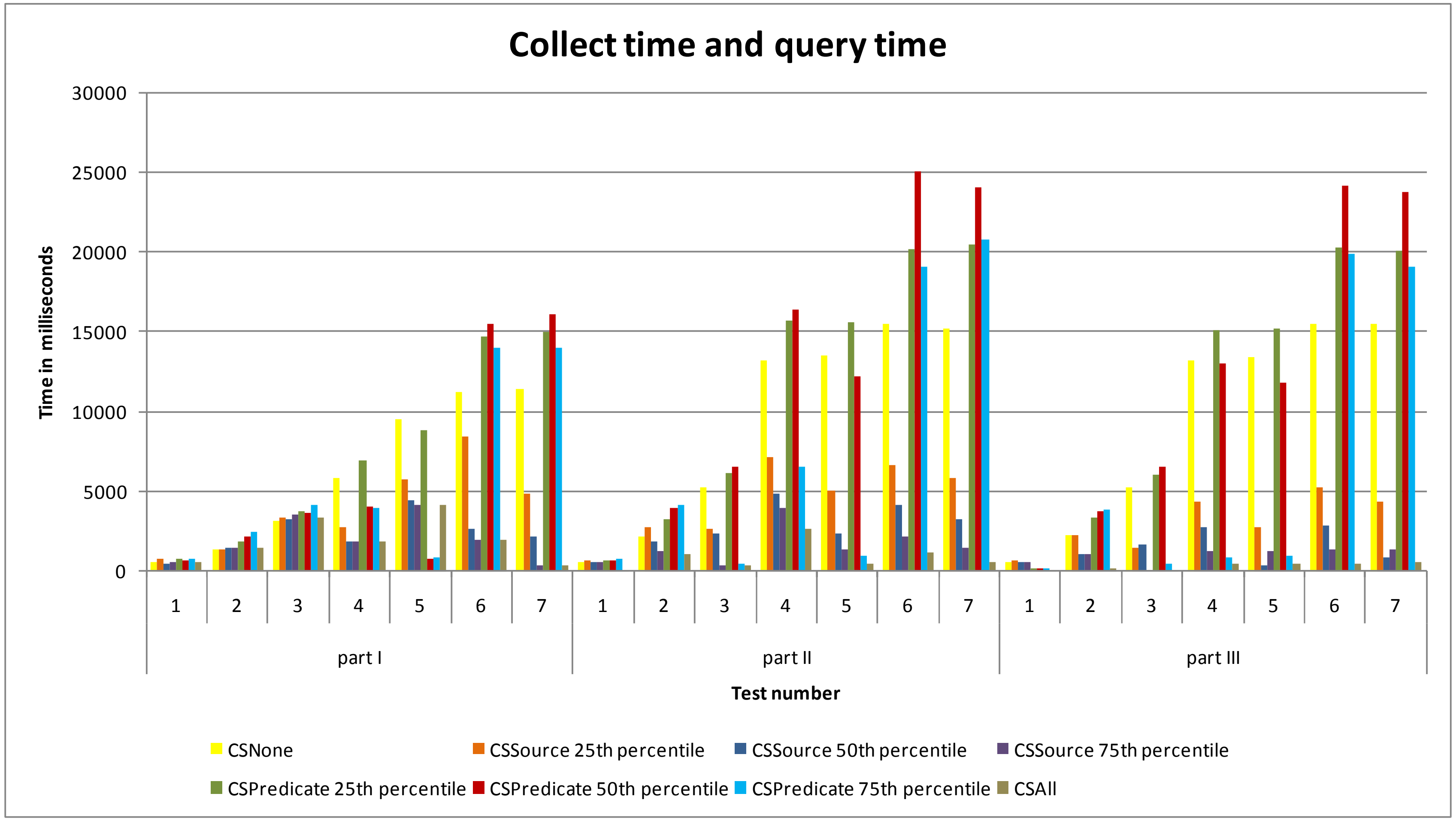


Chart 17: Collect time and query time

To conclude, all the evaluation criteria have shown us that CacheStrategySource performs much better than CacheStrategyPredicate; Chart 17 indicates this as well. Chart 18 illustrates in more detail the collect time and query time for CacheStrategySource (each three of the sizes). It shows us that 75th percentile has the best performance; however, the cache size is 75th percentile and is thus quite large. The 50th percentile also performs well, while having a much smaller size; actually, in many cases its performance is close to the performance of the 75th percentile (e.g., part I query 4 and 5, part II query 5). Consequently, the 50th percentile cache size seems to offer the best balance between cache size and performance.

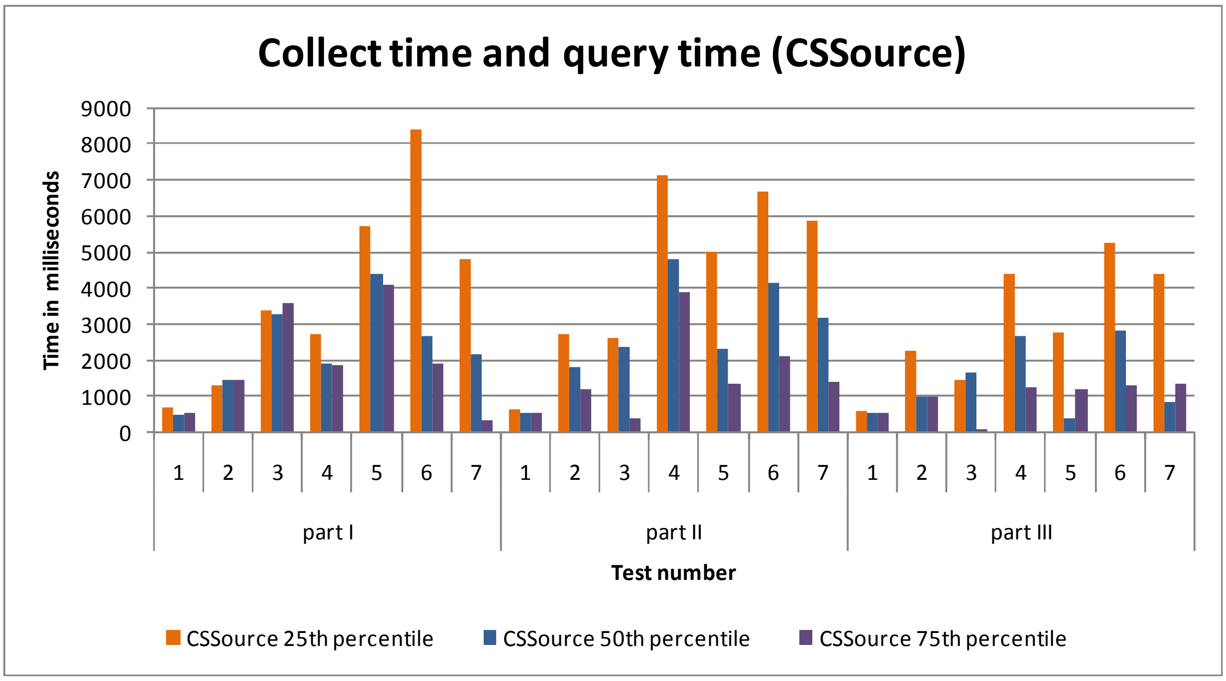


Chart 18: Collect time and query time CacheStrategySource

It can also be observed that CacheStrategySource's performance is still much lower than that of CacheStrategyAll, the hypothetical best-case scenario. It considered future work to investigate how this cache strategy can be optimized to lessen the gap with CacheStrategyAll. .

Chapter 9 Future Work

This chapter describes what extensions and improvements can be made to the work done in this thesis. The first and second sections describe future work concerning the Cache and Source index strategies, respectively. The last section describes the possibility of a query mechanism, which allows queries to be resolved both locally and remotely.

9.1 Source Index Strategy

We have implemented and analyzed two source index strategies, used to determine the relevant sources of a particular query. The mechanism needed to be as selective as possible while still keeping the size of the Source Index Model minimal. Several other source index strategies can be designed, they can vary along two dimensions: what kind of summary information is stored in the Source Index Model (e.g., range of a predicate, number of triples that use a predicate), and how this information is represented by RDF triples.

The first Source Index Strategy, called SourceIndexStrategy1, maintained in the summary model for each source what predicates and corresponding domains it uses. A major benefit for this approach is that is very selective however in some particular case, it is too selective. It is considered future work to investigate the impact of this flaw and how it can be solved.

The query analyzing component, responsible for identifying the concrete predicates used in the query, currently ignores the filter clause of a SPARQL query. However, this clause can also link variables to concrete predicate URIs, using the sameTerm() function; as a result, some sources relevant to the query could (wrongfully) be ignored. One of the most important issues for the future is to extend the query analyzing component to also take into account the filter clause.

Currently the Source Index Model is stored in memory and has no size limitations. Consequently, it would be interesting to develop a kind of caching mechanism for the Source Index Model itself. Furthermore, the freshness of the data in the Source Index Model is not taken into account; in other words, it is assumed that a discovered source never changes once its summary information has been added. This is obviously an unrealistic assumption, and it will be necessary to sporadically update the source index information of a particular source.

Finally, alternative formats can be investigated for storing the summary information present in the Source Index Model. Currently the Source Index Model is being stored as an RDF Graph; alternatively, an SQL lightweight database could be employed to try and improve the performance, decrease the space usage and facilitate updating the Source Index Model. The algorithm responsible for creating the Source Index Query, a query which can be posed to the

Source Index Model to determine the relevant sources for a query, is flexible enough to support other types of queries (e.g., SQL SourceIndexQuery) to pose to the Source Index Model.

9.2 Cache Strategy

In this section, we investigate extensions and improvements for the chosen eviction strategy and caching mechanism.

Until now, one cache eviction strategy has been implemented, namely Least Recently Used (LRU). However there exist other eviction strategies that can also be employed. An example of such an eviction strategy is Adaptive Replacement Cache (ARC). This eviction strategy maintains two lists, one which contains elements that were accessed only once recently; and one which contains elements that are accessed more than once recently. The order of the elements in the lists is determined by recentness, elements that were least recently used are first removed first from the list. Depending on the size and the least recently used elements of both lists the algorithm will determine of which list the least recently used element should be removed. ARC is an algorithm which avoids the drawbacks from both LRU and Least Frequently Used (LFU), an algorithm which evicts the least frequently used element. By treating the elements that are accessed only once recently separately, it avoids that the elements that are only accessed once recently remove elements that are recently accessed many times. By also taking into consideration the recentness, ARC avoids those elements that were accessed many times in the past stay in the cache. As mentioned before LRU is an eviction strategy which scores best in situations where recentness plays a major role. ARC also performs well in such situations, with as benefit that it prefers the elements that are accessed many times recently over the elements that are only accessed once recently. A drawback of ARC is that it causes more overhead than the simpler LRU.

Additionally to implementing a known eviction strategy, a variant or a completely new strategy can be designed to better fit our purposes. The currently implemented LRU eviction strategy can also be optimized, especially the algorithm for removing aged elements from the cache. One possible optimization is to use a tree structure to maintain the case elements in order to optimize the remove aged elements algorithm.

In case several eviction strategies are available, the choice of which eviction strategy needs to be used could be left to the designer of the mobile application. Alternatively, an optimized implementation of Cache Strategy could dynamically choose at runtime which eviction strategy is more suitable for the current situation.

For now, two cache strategies have been implemented and analyzed: Cache Strategy source, which caches entire RDF sources that have been used previously in resolving a query; and Cache

Strategy predicate, which caches triples with predicates that were used in previous queries. It can be noted that both of these strategies cache the RDF information needed for solving query. However query results themselves could be cached as well; this could lead to a whole new set of cache strategies that can be designed, implemented and compared.

Another topic is what type of elements is stored in the cache. For the first cache strategy an entire RDF sources is considered as one cache element and is stored as an RDF Graph in the cache. The second cache strategy sees all the RDF triples using a specific predicate as one cache element and stores the RDF triples as a list structure in the cache. An idea is to maintain the RDF information in another data structure to store it in the cache.

Currently it was assumed that the sources offered by the different entities do not change over time, off course this assumption is faulty and the cache strategies will need to change in order to support the update of the sources. This could be done by determining a max time period that a cache element can stay in the cache without updating the source.

Finally the location of the cache can be considered; currently the cache strategies keep the cache entirely in memory. However it would be interesting to investigate a cache strategy which stores the cache partially on the disk. The idea here is to have a cache that is stored partially in memory and partially on disk. When an element is removed from the cache in memory, it will be added to the cache on disk.

9.3 Querying on the Mobile device

Until now, a remote query service has been used to query an RDF Graph and to analyze, and parse SPARQL queries. In the future, it will become possible to query RDF graphs and analyze SPARQL queries on the device itself, (i.e., on the client side of the SCOUT framework). However this does not mean that the remote query service should vanish; instead this service offers the opportunity to the client to process SPARQL queries either locally or remotely. In cases where the mobile device is slow (e.g., a slow processor), delegating the query resolving to a remote service can be advantageous; on the other hand, in case of a slow (and expensive) network connection, it can be better to handle the query locally. Consequently a kind of mechanism that determines at runtime where a query is solved (remotely or locally) in order to provide maximum flexibility and performance can be developed. This mechanism will have to take into account factors such as device capabilities, network connectivity, etc.

Chapter 10 Conclusion

This thesis has shown how a source index model and caching mechanism can be used to reduce the amount of sources that need to be downloaded when collecting the relevant data for a given query, and to speed up the querying process itself. Experiments were conducted to validate these mechanisms, and to compare several variants to each other. This final chapter summarizes the benefits and drawbacks of the developed mechanisms for maintaining the source index model and caching the encountered sources.

10.1 Source index strategy

The first mechanism, responsible for maintaining the source index model, allows us to determine which sources contain relevant information to solve a particular query. Two variants of this mechanism (called source index strategies) were investigated, which differ in the type (and amount) of information that is stored in the source index model.

The first one, called Source Index Strategy 1, maintains for each source what predicates are used together with their "actual" domains, i.e., the types of resources that occur as subjects of the predicate in the source. The second one, called Source Index Strategy 2, has a source index model that only contains information of the used predicates per source.

As discussed and investigated in the evaluation chapter, Source Index Strategy 2 is much quicker than Source Index Strategy 1 despite the fact that Source Index Strategy 1 is more selective (i.e., find less relevant sources) than Source Index Strategy 2. The underlying reason is that the source index model for Source Index Strategy 1 is much larger than for Source Index Strategy 2 since it contains much more information, and that it needs a more complex Source Index Query since the query must also consider the domain. Additionally, the first strategy can give rise to false-negatives; i.e., sources that are wrongfully excluded from the set of relevant sources (see section 5.4.1).

It can therefore be concluded that Source Index Strategy 2 is the better strategy, as the increase in selectivity of Source Index Strategy 1 does not outweigh its drawbacks related to querying speed and size.

10.2 Cache Strategy

The second mechanism developed in this thesis is a caching mechanism for the encountered data sources. This mechanism decides which data from the encountered sources is kept locally, in order to avoid having to download the relevant sources (as identified by one of the strategies mentioned above) every time a query is issued to the Environment Model.

Two main caching strategies were investigated in detail. The first one is CacheStrategySource, which caches entire sources, while the second one CacheStrategyPredicate only caches triples which employ a specific predicate. Since the granularity of CacheStrategyPredicate is much smaller, the resulting cache is five to eight times smaller than the one for CacheStrategySource. This decreased granularity also leads to a smaller set of data on which the given (original) query needs to be executed, as only specific triples using a predicate from the given query are returned, as opposed to all triples from sources containing a predicate from the given query. Therefore, the execution time of the given query is minimized.

However, if everything is taken into consideration (i.e., download time, composition time of the relevant data and execution of the query on the relevant source), CacheStrategySource is actually a lot quicker than CacheStrategyPredicate. The underlying reason is that CacheStrategySource has a smaller overhead and needs to download the sources a lot less than CacheStrategyPredicate. Therefore, CacheStrategySource is able to quicker compose the relevant data on which the query is being executed, despite the fact that CacheStrategyPredicate returns a smaller set of relevant data.

We can therefore conclude that CacheStrategySource is the better cache strategy, despite the fact that it needs more space and executes the query on a larger set of relevant data and that it needs much more memory. It is considered future work to further investigate the cache strategies, how they can be optimized and to invent and implement new cache strategies.

Bibliography

[1] Sven Casteleyn, Olga De Troyer William Van Woensel, A Framework for Decentralized, Context-Aware Mobile Applications Using Semanctic Web technology.

[2] Matthew Fisher, Ryan Blace, Andrew Perez-Lopez John Hebeler, *Semantic Web Programming*. Indianapolis, Indiana: Wiley Publishing, 2009.

[3] Beat Signer. (2009, Nov.) Web Information Systems: The Semantic Web.

[4] (2007, Juni) en.wikipedia.org. [Online]. http://en.wikipedia.org/wiki/File:W3c-semantic-web-layers.svg

[5] W3C. (2004, Feb.) W3C.org. [Online]. http://www.w3.org/TR/2004/REC-owl-features-20040210/#s1.3

[6] (2008, Januari) W3C. [Online]. http://www.w3.org/TR/rdf-sparql-query/

[7] (2009, November) Jena. [Online]. http://jena.sourceforge.net/

[8] eHow. (2010, April) Cache. [Online]. http://www.ehow.co.uk/cache/

[9] Caswell D. Debaty P., "Uniform Web Presence Architecture for People, Places and Things.," *Personal Communications*, vol. Issue 4, no. Volume 8, pp. 46-51, 2001.

[10] Barton John Kindberg Time, "A Web-based nomadic computing system," *Computer Networks*, vol. vol 35, no. 4, pp. 443-456, 2001.

[11] Kindberg Tom Barton John, "The Cooltown User Experience," in *CHI 2001 Workshop: Building the Ubiquitous Computing User Experience.*, 2001.

[12] Langheinrich M. Roduner C., "Publishing and Discovering Information and Services for Tagged Products," in *19th International Conference on Advanced Information Systems Engineering*. Trondheim, Norway: Springer Berlin / Heidelberg, 2007, pp. 501-515.

[13] Jones Joel Tummala Harsha, "Developing spatially-aware content management systems for dynamic, location-sepific information in mobile systems," in *3rd ACM international workshop on Wireless mobile applications and services on WLAN hotspots, Mobility support and location awareness*. Cologne, Germany: ACM, 2005, pp. 14-22.

[14] Vazques J.I., Abaitua J. Lopez-de-Ipina D., "A Context-aware Mobile Mash-up Platform For Ubiquitous Web," in *3rd IET International Conference on Intelligent Environments*. Ulm, Germany:

ACM, 2007, pp. 116-123.

[15] Rossi Gustavo, Gordillo Silvia, De Cristofolo Valeria Choalliol Cecilia, "Designing and Implementing Physical Hypermedia Applications," in *ICCSA 2006, UWSI 2006*. Heidelber Berlin: Springer, 2006, pp. 148-157.

[16] Persson Per, Sandin Anna, Nystrom Hanna, Cacciatore Elenor, Bylund Markus Espinoza Fredrik, "GeoNotes: Social and Navigational Aspects of Location-Based Information Systems," in *Ubicomp 2001: Ubiquitous Computing*. Heilderberg, Germany: Springer, 2001, pp. 2-17.

[17] Hungkeng Pung, Paulito P. Palmes, Toa Gu Wenwei Xue, "Schema matching for context-aware computing," in *10th international conference on Ubiquitous computing*. Seoul, Korea: ACM, 2008, pp. 292-301.

[18] Steenkiste Peter Judd Gleen, "Providing Contextual Information to Pervasive Computing Applications," in *1st IEEE International Conference on Pervasive Computing and Communications*. Fort Worth, Texas, USA: IEEE Computer Society, 2003, pp. 133-142.

[19] Dominic Battre, "Caching of intermediate results in DHT-based RDF stores," *International Journal of Metadata, Semantics and Ontologies*, vol. Vol 3, no. Issue 1, pp. 84-93, January 2008.

[20] Heiner Stuckenschmidt, "Similarity-Based Query Caching," in *Flexible Query Answering Systems*. Amsterdam: Springer Berlin / Heidelberg, 2004, pp. 295-306.

[21] Michael J. Franklin, Bjorn T. Jonsson, Divesh Srivastava, Michael Tan Shaul Dar, "Semantic Data Caching an Replacement," in *Proceedings of the 22th International Conference on Very Large Data Bases*. Maryland, United States: Morgan Kaufmann Publishers Inc. , 1996, pp. 330-341.

[22] Harrick M. Vin, Asit Dan, Dinkar Sitaram Renu Tewari, *Resource-based Caching for Web Servers*. Austing, Hawthorne: The University of Texas, T.J. Watson Research Center, 1998.

[23] David Karger Dennis Quan, *How to Make a Semantic Web Browser*. Cambridge: T.J Watson Research Center, MIT CSAIL, 2004.

[24] Ulf Leser Bastian Quilitz, *Querying Distributed RDF Data Sources with SPARQL*. Berlin: Humboldt-Universitat, 2008.

[25] Richard Vdovjak, Geert-Jan Houben, Jeen Broekstra Heiner Stuckenschmidt, "Index structures and algorithms for querying distributed RDF repositories," in *International World Wide Web Conference*. New York, NY, USA: ACM, 2004, pp. 631-639.

[26] Richard Helm, Ralph Johnson, John M. Vlissides Erich Gamma, *Design Patterns: Elements of*

Reusable Object-Oriented Software.: Addison-Wesley Professional, 1994.

[27] Wikipedia. http://en.wikipedia.org/wiki/Ontology_%28information_science%29. [Online]. http://en.wikipedia.org/wiki/Ontology_%28information_science%29

Chapter 11 Appendix

11.1 http://wilma.vub.ac.be/~eparet/elien_foaf_N.rdf (RDF file)

```
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffe <http://www.w3.org/2000/01/rdf-schema#seeAlso> <http://wise.vub.ac.be/members/pieter/> .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffe <http://xmlns.com/foaf/0.1/mbox_sha1sum> "620f0f01e8095af4bcf1a91e9f7f12cbc6c46b2a" .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffe <http://xmlns.com/foaf/0.1/name> "Pieter Callewaert" .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffe <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffe <http://xmlns.com/foaf/0.1/givenname> "Pieter" .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffc <http://www.w3.org/1999/02/22-rdf-syntax-ns#value> "http://www.google.be/ig" .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX8000 <http://www.w3.org/2000/01/rdf-schema#seeAlso> <http://wise.vub.ac.be/members/johny> .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX8000 <http://xmlns.com/foaf/0.1/mbox_sha1sum> "72b9db61f552b422e8d9bac106e063f8ccfea415" .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX8000 <http://xmlns.com/foaf/0.1/name> "Johny Paret" .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX8000 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7fff <http://www.w3.org/2000/01/rdf-schema#seeAlso> <http://wise.vub.ac.be/members/frieda> .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7fff <http://xmlns.com/foaf/0.1/mbox_sha1sum> "8a1f61989d9034574966e8e9430819624d1abd01" .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7fff <http://xmlns.com/foaf/0.1/name> "Frieda Van Severen" .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7fff <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffd <http://www.w3.org/2000/01/rdf-schema#seeAlso> _:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffc .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffd <http://xmlns.com/foaf/0.1/name> "Sofie Callewaert" .
_:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffd <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .

<http://www.google.be> <http://webns.net/mvcb/errorReportsTo> <mailto:leigh@ldodds.com> .
<http://www.google.be> <http://webns.net/mvcb/generatorAgent> <http://www.ldodds.com/foaf/foaf-a-matic> .
<http://www.google.be> <http://xmlns.com/foaf/0.1/primaryTopic> <http://vub.ac.be/elien_paret/#me> .
<http://www.google.be> <http://xmlns.com/foaf/0.1/maker> <http://vub.ac.be/elien_paret/#me> .
<http://www.google.be> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/PersonalProfileDocument> .

<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/mbox_sha1sum> "84db5efb9a3ecebeda27be3957df13c171c4b9e2" .
<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/givenname> "Elien" .
<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/title> "Mr" .
<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/knows> _:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7fff .
<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/knows> _:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffe .
<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/nick> "MyNickName" .

<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/phone> <tel:026293754> .
<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/homepage> <http://wise.vub.ac.be/members/Elien/> .
<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/workplaceHomepage> <http://wise.vub.ac.be/> .
<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/knows> _:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX8000 .
<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/name> "Elien Paret" .
```

```
<http://vub.ac.be/elien_paret/#me> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/schoolHomepage> <http://www.vub.ac.be/> .
<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/knows> _:AX2dX622daeb5X3aX126f4fa52ffX3aXX2dX7ffd .
<http://vub.ac.be/elien_paret/#me> <http://xmlns.com/foaf/0.1/family_name> "Paret" .
```