LarKC

The Large Knowledge Collider
a platform for large scale integrated reasoning and Web-search

FP7 – 215535

D1.2.2
Improved Operational Framework

Coordinator: Gaston Tagni
With contributions from: Gaston Tagni (VUA), Zhisheng Huang (VUA), Stefan Schlobach (VUA), Annette ten Teije (VUA), Frank van Harmelen (VUA), Barry Bishop (UIBK), Florian Fischer (UIBK), Vassil Momtchev (Ontotext), Yi Zeng (WICI), Yan Wang (WICI), Yi Huang (Siemens), Georgina Gallizo (HLRS), Matthias Assel (HLRS), Jose Quesada (MPI)
Quality Assesor: Michael Witbrock (CycEur)
Quality Controller: Reto Krummenacher (UIBK)

Document Identifier:	LarKC/2008/D1.2.2
Class Deliverable:	LarKC EU-IST-2008-215535
Version:	version 1.0.0
Date:	March 30, 2010
State:	final
Distribution:	public

Executive Summary

This deliverable is the second of a series of three deliverables aimed at defining an Operational Framework for scalable reasoning in LarKC and as such is a continuation of the work reported in Deliverable D1.2.1 - Initial Operational Framework (M7). The main contributions of this document are: First, a detailed analysis and discussion of different plug-ins currently being developed in LarKC in the context of the different technical work packages along with a discussion of several re-use possibilities between plug-ins and their sub-components. The second contribution is the specification of a series of design patterns aimed at supporting the development of plug-ins and other components in LarKC to achieve reasoning at Web-scale.

Document Information

IST Project Number	FP7 – 215535	**Acronym**	LarKC
Full Title	Large Knowledge Collider		
Project URL	`http://www.larkc.eu/`		
Document URL			
EU Project Officer	Stefano Bertolo		

Deliverable	**Number**	1.2.2	**Title**	Improved Operational Framework
Work Package	**Number**	1	**Title**	Conceptual Framework

Date of Delivery	**Contractual**	M24	**Actual**	31-March-10
Status	version 1.0.0		final ☒	
Nature	prototype ☐ report ☒ dissemination ☐			
Dissemination Level	public ☒ consortium ☐			

Authors (Partner)	Barry Bishop (UIBK), Florian Fischer (UIBK), Gaston Tagni (VUA), Zhisheng Huang (VUA), Vassil Momtchev (OntoText), Yi Zeng (WICI), Yan Wang (WICI), Jose Quesada (MPI), Yi Huang (SIEMENS), Georgina Gallizo (HLRS), Matthias Assel (HLRS), Stefan Schlobach (VUA), Annette ten Teije (VUA), Frank van Harmelen (VUA)		
Resp. Author	Gaston Tagni (VUA)	**E-mail**	gtagni@cs.vu.nl
	Partner VUA	**Phone**	+31 (20) 59-87753

Abstract (for dissemination)	This deliverable is the second of a series of three deliverables aimed at defining an Operational Framework for scalable reasoning in LarKC and as such is a continuation of the work reported in Deliverable D1.2.1 - Initial Operational Framework (M7). The main contributions of this document are: First, a detailed analysis and discussion of different plug-ins currently being developed in LarKC in the context of the different technical work packages along with a discussion of several re-use possibilities between plug-ins and their sub-components. The second contribution is the specification of a series of design patterns aimed at supporting the development of plug-ins and other components in LarKC to achieve reasoning at Web-scale.
Keywords	LarKC platform, LarKC Plug-ins, Web scale reasoning, Semantic Web

Project Consortium Information

Acronym	**Partner**	**Contact**
Semantic Technology Institute Innsbruck `http://www.sti-innsbruck.at`		Prof. Dr. Dieter Fensel Semantic Technology Institute (STI) Innsbruck, Austria E-mail: dieter.fensel@sti-innsbruck.at
AstraZeneca AB `http://www.astrazeneca.com/`		Bosse Andersson AstraZeneca Lund, Sweden E-mail: bo.h.andersson@astrazeneca.com
CEFRIEL SCRL. `http://www.cefriel.it/`		Emanuele Della Valle CEFRIEL SCRL. Milano, Italy E-mail: emanuele.dellavalle@cefriel.it
CYCORP, RAZISKOVANJE IN EKSPERIMENTALNI RAZVOJ D.O.O. `http://cyceurope.com/`		Dr. Michael Witbrock CYCORP, RAZISKOVANJE IN EKSPERIMENTALNI RAZVOJ D.O.O., Ljubljana, Slovenia E-mail: witbrock@cyc.com
Hchstleistungsrechenzentrum, Universitaet Stuttgart `http://www.hlrs.de/`		Georgina Gallizo Hchstleistungsrechenzentrum, Universitaet Stuttgart Stuttgart, Germany E-mail : gallizo@hlrs.de
Max-Planck-Institut fur Bildungsforschung `http://www.mpib-berlin.mpg.de/index_js.en.htm`		Dr. Lael Schooler, Max-Planck-Institut fr Bildungsforschung Berlin, Germany E-mail: schooler@mpib-berlin.mpg.de
Ontotext Lab, Sirma Group Corp. `http://www.ontotext.com/`		Atanas Kiryakov, Ontotext Lab, Sirma Group Corp. Sofia, Bulgaria E-mail: atanas.kiryakov@sirma.bg
SALTLUX INC. `http://www.saltlux.com/EN/main.asp`		Kono Kim SALTLUX INC Seoul, Korea E-mail: kono@saltlux.com
SIEMENS AKTIENGESELLSCHAFT `http://www.siemens.de/`		Dr. Volker Tresp SIEMENS AKTIENGESELLSCHAFT Muenchen, Germany E-mail: volker.tresp@siemens.com
THE UNIVERSITY OF SHEFFIELD `http://www.shef.ac.uk/`		Prof. Dr. Hamish Cunningham THE UNIVERSITY OF SHEFFIELD Sheffield, UK E-mail: h.cunningham@dcs.shef.ac.uk
VRIJE UNIVERSITEIT AMSTERDAM `http://www.vu.nl/`		Prof. Dr. Frank van Harmelen VRIJE UNIVERSITEIT AMSTERDAM Amsterdam, Netherlands E-mail: Frank.van.Harmelen@cs.vu.nl
THE INTERNATIONAL WIC INSTITUTE, BEIJING UNIVERSITY OF TECHNOLOGY `http://www.iwici.org/`		Prof. Dr. Ning Zhong THE INTERNATIONAL WIC INSTITUTE Mabeshi, Japan E-mail: zhong@maebashi-it.ac.jp
INTERNATIONAL AGENCY FOR RESEARCH ON CANCER `http://www.iarc.fr/`		Dr. Paul Brennan INTERNATIONAL AGENCY FOR RESEARCH ON CANCER Lyon, France E-mail: brennan@iarc.fr

Table of Contents

1 Introduction

This deliverable is the second of a series of three deliverables aimed at defining an Operational Framework for scalable reasoning in LarKC and as such is a continuation of the work reported in Deliverable D1.2.1 - Initial Operational Framework (M7). The goal of this deliverable is twofold: Firstly, to briefly report on the current status of the larKC platform, which has been improving incrementally over the last seventeen months. To this end we give a short overview of the functionality provided by the main components of LarKC, namely its plug-in types and data layer component. Secondly, to report on the results of a series of activities aimed at improving re-use and tighter integration of components in LarKC. The early architecture of LarKC exploited a separation of components along conceptual lines, i.e. identify, transform, select, reason and decide. We showed how existing software components could be fitted conveniently into these categories. Their convenience was manifested in the quick delivery of interesting demonstrators and large set of plugins, often wrapping third-party solutions.

However, a well-known design trade-off in in computer science is between modularity and efficiency. The purpose of this deliverable is to investigate a closer coupling between components.

Contributions of this Deliverable. The main contributions of this deliverable are the following: First, a detailed analysis and discussion of different plug-ins currently being developed in LarKC in the context of the different technical work packages. This includes a description of the plug-ins' functionality, an overview of the plug-ins' architecture along with a discussion of several possibilities for re-use between plug-ins and their sub-components. The second contribution of this deliverable is the specification of a series of design patterns aimed at supporting the development of plug-ins and other components in LarKC to achieve reasoning at Web-scale. In the scope of WP5 The Collider Platform, several design patterns have been identified as a result of the analysis of performance and scalability of the LarKC platform architecture and current prototype implementations. These design patterns are modelling the features of different kind of LarKC workflows, which may present some limitations on performance and scalability, under certain deployment conditions, and specifying concrete requirements with regards to performance and scalability improvement.

Structure of this document. This document is structured as follows: Chapter 2 discusses the relation between scalability, performance and tighter integration of components in LarKC. Chapters 3, 4 and 5 are concerned with reusability of plug-ins and their components. In particular, Chapter 3 is concerned with discussing different re-use possibilities from the perspective of the Select and Identify plug-in types. Chapter 4 is devoted to the discussion of re-use possibilities w.r.t Transform plug-ins. Finally, Chapter 5 is devoted to discussing re-use from the perspective of Reason plug-ins. Chapter 6 introduces five design patterns aimed at supporting developers in the design of plug-ins and workflows in LarKC. The last chapter of this document, Chapter 7 concludes the document.

2 Improving Performance by Tighter Integration of Components

2.1 Integration of Selection and Query

In what regards to selection strategies, we have conducted a comparison of user interests-based query refinement and the integration of interests-based selection and querying. In Chapter 3 of D2.3.2 [26] and Chapter 4 of D4.3.2 [16], we reported some experiments on the integration of selection and querying. Here we give a brief conclusion of the results and their meaning w.r.t scalability on LarKC. The reader is referred to the aforementioned deliverables for further details.

The evaluation performed in Deliverable D2.3.2 is based on three different query strategies, namely:

- Strategy 1: Selection based on the original query. This strategy uses the original query posed by the user without applying any query refinement mechanism.
- Strategy 2: Interests-based query refinement. The original query posed by the user is refined (rewritten) based on the user's interests. The refined query is then used to select a subset of the dataset.
- Strategy 3: Querying with Interests-based selection. This strategy selects a relevant subset of the dataset based on the user's interests, which are specified using the context parameter of the Select plug-in type). After this, the original query posed by the user is answered w.r.t the subset of the dataset computed before.

Experimental results showed that since the "Interests-based Query Refinement" strategy takes more constraints in comparison to the original query posed by the user, it requires more processing time for answering the query. In fact, results showed that as the size of the data increases the time to answer the query increases very rapidly; which suggests that this method does not scale well w.r.t the answering time. Although this method takes more time to process a given query, as reported in D2.3.1 [27], the quality of the results is much better than the quality of results obtained without query refinement.

Strategy 3 integrates selection and querying in order to provide a scalable method for query processing on large scale data. Since this strategy selects relevant subsets of a dataset in advance, the required query time reduces a lot, and as the size of the dataset grows the query time is always less and does not increase equally fast in comparison to strategies 1 and 2. The quality of results is the same as that of strategy 2. In summary, this strategy scales better than the previous two.

Although the selection and querying process can benefit from user interests as contextual constraints, we find that from the processing time perspective, if we add more constraints for the query, more processing time may be needed. This is both true for both strategies 2 and 3. Hence, the balance between contextual constraints and processing time should be considered.

In conclusion, by integrating selection and query answering, query time can be significantly reduced (assuming the user does not require a complete set of answers and prefers results that are much more related to her background).

2.2 Interleaving Reasoning and Selection

Interleaving reasoning and selection is considered to be an approach to improve the performance of the LarKC platform. The main idea of the interleaving framework is to use selectors to select only limited and relevant part of data for reasoning, so that the efficiency and the scalability of reasoning can be improved. The general scenario of interleaving reasoning and selection consists of the following three steps:

- **Selection**: Use a selector to select parts of the input data.
- **Reasoning**: Use a reasoner to reason over the selected data;
- **Deciding**: Use a decider to decide whether the procedure should be stopped in order to return the answers to the user or, the process should go back to the selection step to continue the interleaving processing.

Several strategies have been proposed for selecting parts of a given dataset. Selection of parts of a dataset can be made based on some relevance measure. For example, a syntactic relevance measure is one that is based on checking symbolic co-occurrances. Alternatively, semantic relevance measures use some kind of background knowledge to provide information about the meaning of the data. The selection can also be made based on the user's preferences or interests. In LarKC, we have developed several approaches for interleaving selection and reasoning including the following (the interested reader is referred to Deliverable D4.3.1 for further details):

- **Query-based Selection**. Query-based selection is a selection strategy that selects data/axioms by examining the relevance of axioms w.r.t. the reasoning query. This selection strategy has been implemented in PION, a system for reasoning with inconsistent ontologies that interleaves reasoning and query-based selection. PION uses selection functions which uses either syntactic or semantic relevance measures for comparing data with the reasoning queries. A detailed description of the PION system is provided in Section 5.4.
- **User Interest-based Selection**. User interest-based selection is a strategy in which selection is made based on the user's interests. This kind of selection is examined with respect to different user interests specified at different granularity levels. In D4.3.1, we investigate Web scale reasoning from the perspective of granular reasoning, and develop several strategies for reasoning at Web scale that consider users' interests at several levels of granularity.
- **Language-based Selection**. Language-based selection is one in which subsets of a dataset are computed by selecting parts of the ontological vocabulary. In D4.3.1, we developed several approaches for anytime instance retrieval by ontology approximation where the approximate ontologies are computed using different subsets of the original ontology's vocabulary. We also report on the results of different experiments using a comprehensive set of realistic ontologies.

A framework for interleaving selection and reasoning requires a close integration between select and reason plug-ins. Both select and reason plug-ins must be implemented in such a way to allow the reason component to give feedback to the select component on the quality of the selection. Based on this feedback and on the previous selection, a select plug-in should be able to make new selections of data that satisfy the needs of the reasoner (or any other plug-in using the results of the selection process). This tight collaboration between select and reason plug-ins should be designed to improve the efficiency of reasoning. That kind of the improvement may also rely on a decider to provide some necessary information for the seleter to make a better selections for the next round of the interleaving process. In Chapter 5 we are going to discuss various scenarios where interleaving reasoning and selection may improve the performance of the LarKC platform.

2.3 Tighter integration with the Data Layer

The data layer is a core component of the LarKC platform that ensures the efficient persistence and communication of information (represented as RDF) among the different plug-ins. This component extends the ORDI Second Generation (SG) framework and adds an abstraction level that hides the physical location of data. Thus, the platform supports greater flexibility in the deployment of complex workflow scenarios and automates the information exchange by value or reference. The extra abstraction layer could also lead to inefficiency related to the processing of remote large datasets or to generic and high-level query interface. In this section we will discuss if the tighter integration between the platform plug-ins and the data layer could lead to increased scalability.

A typical example of computation that is tightly coupled with the data is the computation performed by the Select plug-in. A selector is responsible for early filtering of data and for narrowing down the relevant information needed for answering a given query. The typical implementation of such plug-in type is very I/O demanding and often requires extended query support to minimize the traffic between the plug-in and data layer processes. A very simple sandbox example is the task of retrieving from the LDSR service [1] the number of people for which their birth place is known. The following extract of XML code illustrates the SPAQRL query encoding this request.

```
PREFIX dbp-ont: <http://dbpedia.org/ontology/>

SELECT *
WHERE {
     ?Person dbp-ont:birthPlace ?BirthPlace ;
}
```

The data request would require to transfer the complete result set and count its size within the plug-in process, which may be a fairly inefficient operation due to 1) network traffic and 2) the overhead of data serialization and deserialization.

[1] http://ldsr.ontotext.com

In order to optimize this behaviour the underlying ORDI engine exposes a new query construct, which allows getting only the result count:

```
PREFIX dbp-ont: <http://dbpedia.org/ontology/>

SELECT *
FROM <http://www.ontotext.com/count>
WHERE {
     ?Person dbp-ont:birthPlace ?BirthPlace ;
}
```

The previous example illustrates a scenario where in order to achieve efficiency and scalability the light-weight retrieval logic is pushed down to the data layer. This will significantly reduce the cost of data transfers over the network and the serialization overhead on the server. In the case of in-process communication (e.g., within the boundaries of a single virtual machine) the optimisation is still beneficial because the ORDI framework utilizes the internal and highly efficient TRREE model. The overall architecture of the data layer and its underlying components is depicted in Figure 2.1. The Data Layer's API provides a standard set of operations, which hide the physical location of the data. However, in scenarios dealing with high volumes of data and information the processing of remote data may become a serious bottleneck. This would enforce a tighter integration between specific plug-ins and the data layer, which could be achieved using data layer engine extensions. Such extensions are successfully tested in the implementation of DualRDF and RDFPageRank selector plug-ins (see [1] for more details).

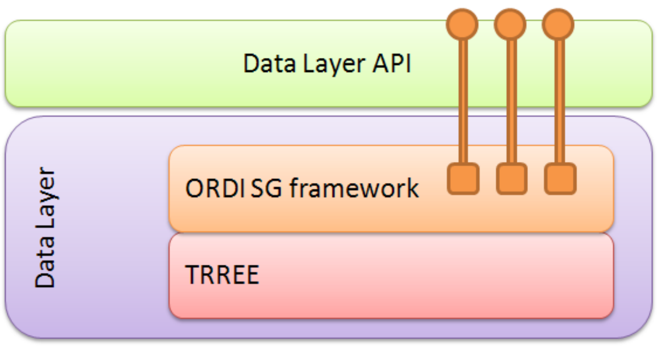

Figure 2.1: Extended functionality supported by LarKC data layer API

3 Select and Identify Plug-in Types

3.1 Introduction

An *Identify* plug-in is a query-driven plug-in whose main role is to identify the subset of data that is relevant for answering a user's query. An *Identify* plug-in is used for narrowing the scope of a reasoning task from all available information sets to those information sets that can contribute to answering a given query. The plug-in's interface defines three input parameters: the user's query that drives the identification/selection mechanism, hence the term query-driven plug-in, a contract parameter that is used for specifying the dimensions of the output and, a context parameter that allows other plug-ins and components in LarKC to access and exchange information about plug-in's state. The plug-in's output is a collection of Information Sets constructed according to the plug-in's contract parameter.

The role of a *Select* plug-in in LarKC is to further narrow the scope or search space of a reasoning task by selecting a subset (or sample) of the data set that has been made available by an instance of the Identify plug-in type. A Select plug-in has three input types: a set of statements representing the data from which samples will be drawn, a contract parameter that is used for defining the dimensions of the output, eg. to specify the size of the sample to be taken and, a context parameter that allows for plug-ins to access and exchange information about the state of the plug-in. Upon execution this plug-in returns a set of statements representing a subset of the set of statements given as input to the plug-in. Although the interface of a Select plug-in seems rather simple is actually quite flexible allowing for different selection techniques and strategies to be implemented as different instances of it. The only requirement is to accept a set of statements and return a (sub)set of statements. For a detailed description of the current architecture of the LarKC platform the reader is referred to Deliverable D5.3.2 - Overall LarKC Architecture and Design [11].

The following sections describe the overall design of some of the Select and Identify plug-ins currently under development in LarKC. The next section will report on the design decisions and overall architecture of a Select plug-ins that take the user's interests into account in order to select parts of a data set.

3.2 Interests-Based Selecter

The users are not always aware that in many cases their query requirements are closely related to their contexts (e.g. their previous interests, etc.) which act as the environmental factors for the query tasks. In addition, the constraints from the contexts do not always explicitly contain in the query input by the users. Following the discussion in D2.3.1 [27], we developed a selecter plug-in named *Interest-Based Selecter*. Its function is to create a selection over the input statements, based on user interests.

Figure 3.1 shows the architecture of the *Interest-Based Selecter*. This plug-in is composed of 5 modules, namely, *Context Manager*, *User Interest Manager*, *Query Manager*, *Query Engine* and *Results Manager*. The *Context Manager* is responsible for managing the context information. The *User Interest Manager* is used to manage user interests information, including the function of extracting user interests. The *Query Manager* is for creating SPARQL queries based on user interests. The *Query Engine* is responsible for executing SPARQL queries. *Results Manager* is used to organize the results of the SPARQL queries.

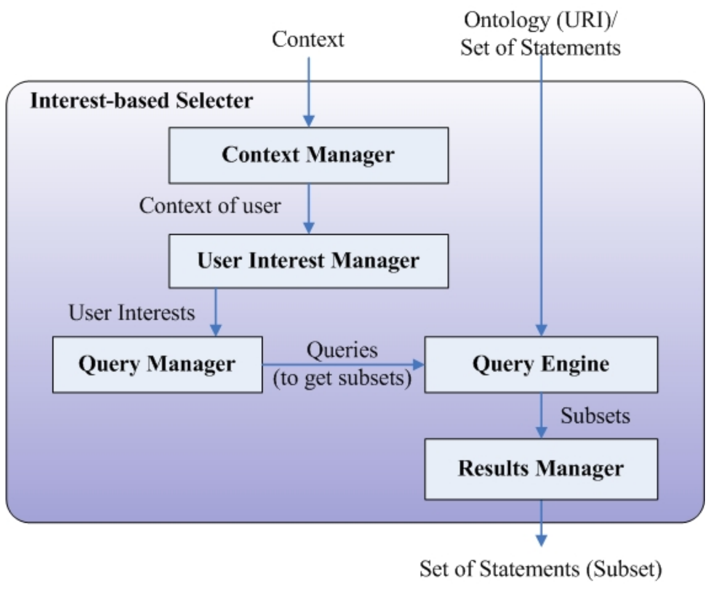

Figure 3.1: The Architecture of the Interest-Based Selecter plug-in

At run time, the plug-in accepts the context parameter and uses the *Context Manager* module to get current user's basic information, such as user identifier (the name or the URI) and the identifier of the data source which includes user interests. Then the *User Interest Manager* module extracts user interests from the given data source with user identifier. The *Query Manager* module is responsible for generating SPARQL queries with constraints driven by the user interests. After that, the *Query Engine* module executes these queries to get a group of subsets. Finally, *Results Manager* module merges them to create a subset of the original dataset, which is relevant to current user's interests.

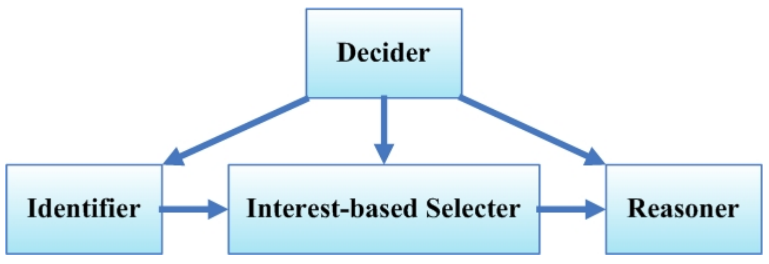

Figure 3.2: The Interest-Based Selecter plug-in in a workflow

Figure 3.2 shows a LarKC workflow that uses the Interest-Based Selecter plug-in. In this workflow, the selecter is the Interest-Based Selecter plug-in which receives output of the identifier as its input, and produces output as the input of the reasoner. Here the identifier can be any identifier plug-ins developed. The reasoner can be any LarKC reasoner plug-ins. The decider interacts with each of the three plug-ins, and it also decides the size of the subset selected by constraints from user interests.

Till now, we have developed two plug-ins, namely, the Interests-Based Selecter and the Interests-Based Reasoner. They are both based on this model. Since our work is user-centric, we need some features to represent a user's basic information. For reuse, we propose to add a set of classes to LarKC API to represent the context of the users, such as user interests. In this way, any plug-in that need to utilize user information could reuse the class provided by the platform.

In the Interests-Based selecter, the class "ContextImpl" could be partially reused. This class implements the interface "Context" provided as LarKC selecter plug-in API. It is responsible to parse the context object. Through the parsing process, the plug-in gets the user information passed by the context, namely, the user name, user interests, etc. So, if any other plug-ins need this kind of user information, they could reuse this class in the Interest-Based selecter. For example, in the Interest-Based Reasoner plug-in that we developed, we have reused the "ContextImpl" class. Both of these plug-ins are available at http://www.wici-lab.org/wici/wiki/index.php/LarKC .

3.3 Random Indexing-based Selecter

The ESA and random indexing (RI, [29]) plugins are selectors. They are used for subsetting a large knowledge space. They take a SPARQL query plus set of rdf triples and produce a smaller set of rdf triples. For brevity, we will describe here RI only. Note that these methods can act on both the plain text description of the concepts from Wikipedia and on RDF molecules.

Random Indexing is a machine learning technique that offers the mechanisms to measure the semantic distance between two "passages" (where a passage is simply defined as a collection of one or more words, or an RDF molecules) in a "semantic space", which is created from the input text. The space is generated by first constructing a matrix whose cell entries are the number of times a given word appears in a given passage. As an example, consider the passage "I have one sister and one brother." If row i corresponds to the aforementioned passage and column j corresponds to the word "one", then the value in cell ij would be 2 because "one" appears in the passage twice. The other words appear only once, so the values in the other cells in row i would be 1.

The space can be viewed in an instructive but oversimplified way: Passages with similar semantic content are located near one another in the space, and words that have occurred in similar passages are also located near one another in the space. This view uncovers one of the benefits of dimension reduction words that did not co-occur, but occurred in similar contexts (e.g., "doctor" and "physician"), will be grouped together as well. Given that the words and passages in the space are arranged according to their semantic content, it is therefore termed a semantic space.

In our particular implementation, Random Indexing starts with a matrix of words by contexts where a context is a Wikipedia concept (we preselected the top 1M concepts to save computation). Then each word and each context is first assigned a random high-dimensional sparse vector: they contain a small proportion of +/- elements (this proportion is called the seed value) with all other elements set to zero. This is enough to make vectors different from each other.

The plain text RI plugin is trained on Wikipedia, and thanks to the paralelism to DBpedia, we can assign weights to nodes in the RDF graphs. The weights are proportional to the semantic similarity between the query and the elements according to RI. The plain text RI plugin needs to convert a SPARQL query into a keywords query. Right now we use the 'extractKeywords' method from the baseline plugin.

Once the sparse binary index vectors are constructed, a word's vector becomes the sum of the vectors for the contexts in which it appears throughout the text corpus. Conversely, a document space can also be constructed as the sum of the index vectors for words appearing in each document. Hence, random indexing does not require a dimensional reduction operation, which is the computationally expensive step that made other methods less than ideal for large scale settings. Random Indexing is based on the fact that a term-document matrix computed from a corpus is sparse. The sparsity is large enough that the vector representations can be projected onto a basis comprising a smaller number of randomly allocated vectors. Due to the sparseness condition, the basis of random vectors has, in general, a high probability of being orthonormal.

The plugins work in three steps, only step 2 is different for RI and ESA (we will discuss the details for each in their own sections).

1. **Selection of the most important Wikipedia concepts**. We followed the procedure in [10]. Although Wikipedia has currently almost four million articles, not all of them are equally useful for statistical semantics. Some articles correspond to too specific concepts (e.g., the game 'Voyage Century Online'). Other pages are uninteresting because they contain almost no text (e.g., specific dates). We also eliminated articles that are just too short (less that 100 words after stemming and removing function words).
   We also used the Link Structure of Wikipedia to determine which concepts are more prominent. We eliminated all concepts that didn't have at least 5 incoming and outgoing links. We ended up with about 1M concepts.

2. **Creation of the space**. This is a one-off operation, but it is computationally expensive (minutes, sometimes hours). For example, computation of RI with 1000 dimensions takes 20 minutes on an office PC. This operation produces a vector for each of the Wikipedia concepts in the first step. The plugins cannot operate without those vector spaces. The one-off computation does not need to happen on the same computer that runs LarKC. Once computation is done, the vector space take about 3-4 gigabyte of space and can be moved to a different location. It can also be shipped with working plugin code, so an end user needs not to worry about the details of space creation.

3. **Computation of nearest neighbors**. This operation takes place every time we need to use Subsetting with a SPARQL query. We compute the weights that will be assigned to the entire LDSR graph. It takes seconds, but this is a time penalty that we need to add to the time it takes OWLIM to compute the answer.

Step two, the creation of the space is one thing that differentiates these plugins from the rest. A plugin has two parts, a part integrated with LarKC and the computation part. This division of labor is by design. The LarKC plugin is a thin client; no heavy-weight computation takes place there.

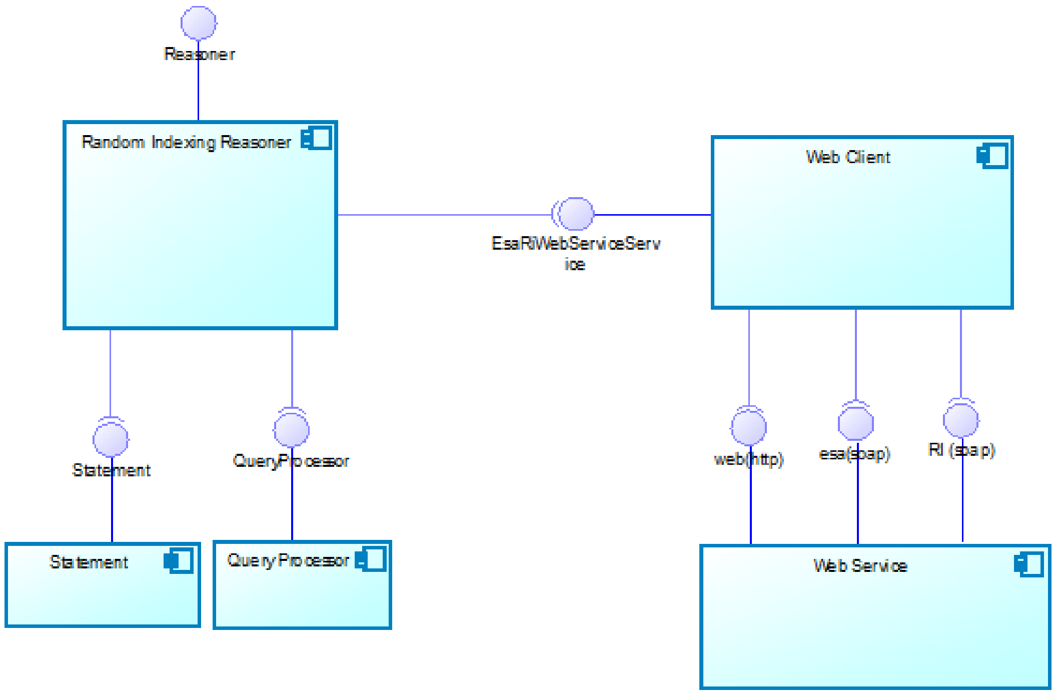

Figure 3.3: The Architecture of the Random Indexing Select plug-in, (implemented as a reasoner)

Figure 3.3 shows the components of the plugin. The left side is the thin client, and the right side is the computationally intensive part implemented as web service. The two parts are separable, the computation code can be called using http and soap, which makes it somewhat reusable without writing specific code. For a (more detailed) class diagram, see D2.3.2. The plugins produce a continuous measure of similarity that can be reused in different ways. For example, it could be used for ontology matching (PION) instead of Google distance, or in any case where approximate matching would be desirable.

4 Transform Plug-in Type

4.1 Introduction

The role of a *Transform* plug-in in LarKC is to transform data from one format to another format. The data to be transformed can be either a query or an information set that is considered relevant for achieving a particular task, e.g. answering a user's query. To support this the LarKC platform defines two types of *Transform* plug-ins, one for transforming queries and the other for transforming information sets. Similar to other plug-in types in LarKC both plug-in types accept a parameter that specifies the contract to be agreed upon by the caller and the plug-in and a parameter for encapsulating the plug-in's context information. The former is usually used for defining the dimensions of the plug-in's output and in the *Transform* plug-ins here described it is used also for setting up the environment of the transformation, e.g. definition of the population, feature pruning threshold and learning algorithm specific parameters, while the latter allows plug-ins to communicate their state to other components in the platform.

This chapter reports on the work related to information set *Transform* plug-ins and specifically transform plug-ins that use machine learning techniques to transform data. The chapter is organized as follows: First, we introduce the machine learning approach *statistical unit node set* (SUNS) which is suitable for the challenging data situation on the Semantic Web. Then we describe the architecture and the functionalities of ML-based *Transform* plug-ins we have developed so far in context of LarKC. Finally we analyze the reusability of components in perspective of other plug-in types.

4.2 ML-based Transformer

4.2.1 Machine Learning Approach SUNS on Semantic Web Data

One of the main characteristics of Semantic Web (SW) data is that it is notoriously incomplete. A popular example is the well known friend-of-a-friend data set where, for privacy concerns and other reasons, some members document exhaustive pri-

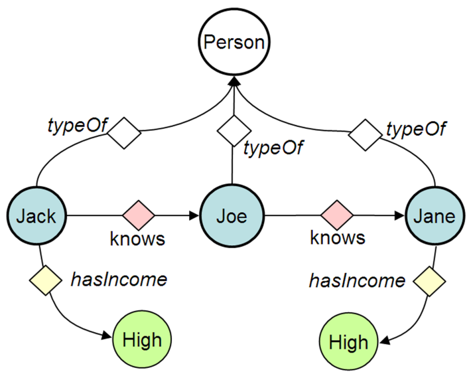

Figure 4.1: Example of an RDF graph displaying a social friendship network

vate and social information whereas almost nothing is known for other members. Although deductive reasoning can be used to complement factual knowledge based on the ontological background, still a tremendous number of potential statements remain to be uncovered. Therefore we are focused on the prediction of potential relationships and attributes by exploiting regularities in the data using statistical relational learning algorithms. An important goal of learning approaches and plug-ins in the WP3 is to estimate the truth values of triples exploiting patterns in the data. Here we need to take into account the nature of the SW which is dynamically evolving and quite noisy. Thus flexibility and ease of use are preferred properties if compared to highly sophisticated approaches that can only be applied by a small number of machine learning experts.

Looking at the data situation on the Semantic Web, there are typically many possible triples associated with an entity (these triples are sometimes called entity molecules or, in our work, statistical unit node set) of which only a small part is known to be true. Due to the large degree of sparsity of the relationship data in the SW, multivariate prediction is appropriate for SW learning. The rows in the learning matrix, i.e. data points, are defined by the key entities (statistical units) in the sample. The columns are formed by nodes that represent the truth values of triples that involve the statistical units. Nodes representing aggregated information form the inputs. The size of the training data set is under the control of the user by means of sampling. Thereby the data matrix is typically independent or only weakly dependent on the overall size of the SW and in consequence the time consume and feasibility of model training is essentially independent of the overall size of the SW.

Defining the Sample

We must be careful in defining the statistical unit, the population, the sampling procedure and the features. A statistical unit is an object of a certain type, e.g., a person. The population is the set of statistical units under consideration. In our framework, a population might be defined as the set of persons that attend a particular university. For learning we use a subset of the population. Based on the sample, a data matrix is generated where the statistical units in the sample define the rows.

The Random Variables in the Data Matrix

We now introduce for each potential triple a *triple node* drawn as a diamond-shaped node in Figure 4.1. The figure shows an example of an RDF graph displaying a social friendship network in which the income of a person is an attribute. Resources are represented by circular nodes and triples represented by labeled directed links from subject node to object node. The diamond-shaped nodes stand for random variables which are in state *one* if the corresponding triples exist. Nodes representing statistical units (here: *Persons*) have a darker rim. A triple node is in state *one* (*true*) if the triple is known to exist and is in state *zero* (*false*) if the triple is known not to exist. Graphically, one only draws the triple nodes in state *one*, i.e., the existing triples.

We now associate some triples with statistical units. The idea is to assign a triple to a statistical unit if the statistical unit appears in the triple. Let's consider the statistical unit *Jane*. Based on the triples she is participating in we obtain $(X, typeOf, Person)$, $(Joe, knows, X)$, and $(X, hasIncome, High)$ where X is a variable that represents a statistical unit. The expressions form the random variables (outputs) and define columns in the data matrix. By considering the remaining statistical units *Jack* and *Joe* we generate the expressions (columns), $(X, knows, Jane)$, $(Jack, knows, X)$. We will not add $(Jane, knows, X)$ since Jane considers no one in the data base to be her friend. We iterate this procedure for all statistical units in the sample and add new expressions (i.e., columns in the data matrix), if necessary. Note that expressions that are not represented in the sample will not be considered. Also, expressions that are rarely true (i.e., for few statistical units) will be removed since no meaningful statistics can be derived from few occurances. In [34] the triples associated with a statistical unit were denoted as *statistical unit node set* , abbreviated as SUNS.

Non-random Covariates in the Data Matrix

The columns we have derived so far represent truth values of actual or potential triples. Those triples are treated as random variables in the analysis. If machine learning predicts that a triple is very likely, we can enter this triple in the data store. We now add columns that provide additional information for machine learning but which we treat as covariates or fixed inputs.

First, we derive simplified relations from the data store. More precisely, we consider the expressions derived in the last subsection and replace constants by variables. For example, from $(X, knows, Jane)$ we derive $(X, knows, Y)$ and count how often this expression is true for a statistical unit X, i.e. we count the number of friends of person X.

Second, we consider two simple types of aggregated features from outside a SUNS. Consider first a binary triple $(X, knows, Jane)$. If Jane is part of another binary triple, in the example, $(X, hasIncome, High)$ then we form the expression $(X, knows, Y) \wedge (Y, hasIncome, High)$ and count how many rich friends a person has. A large number of additional aggregated features are possible but so far we restricted ourselves to these two types.

After construction of the data matrix we prune away columns which have *ones* in fewer than ϵ percent of all rows or in more than $(1 - \epsilon)$ of all rows, where ϵ is usually a very small number. Thus we remove aggregates features that are very rarely true or almost always true, since for those no meaningful statistical analysis is possible. Note that by this pruning procedure we have reduced the exponential number of random variables to typically a much smaller set.

Algorithms for Learning with Statistical Units Node Sets

A row in the resulting data matrix contains external inputs based on aggregated information (if available) and typically a large number of binary and sparse outputs. A *one* stands for a triple known to be true and a *zero* for a triple whose truth value is unknown. In this situation, multivariate prediction approaches have been most successful [35]. In multivariate prediction all outputs are jointly predicted such that statistical strength can be shared between outputs. The reason is that some or all model parameters are sensitive to all outputs, improving the estimates of those parameters. The approaches we are employing here are based on a matrix

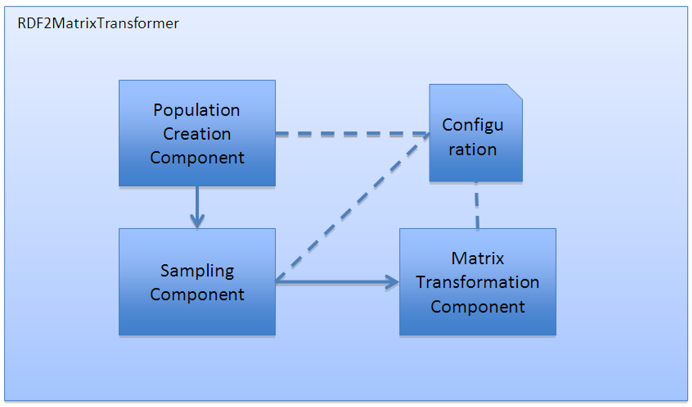
RDF2MatrixTransformer
Population Creation Component
Configu ration
Sampling Component
Matrix Transformation Component

Figure 4.2: Architecture of RDF2Matrix transformer plug-in

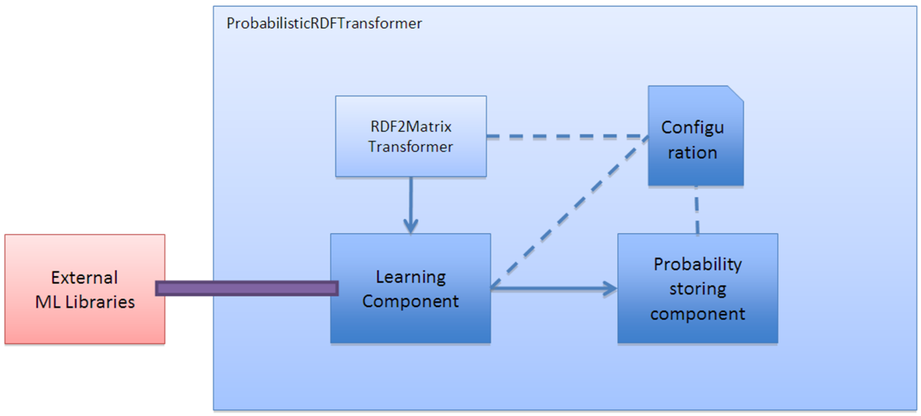

Figure 4.3: Architecture of ProbabilisticRDF transformer plug-in

completion of the complete data matrix, including inputs and outputs.[1] We investigate matrix completion based on a singular value decomposition (SVD), matrix completion based on non-negative matrix factorization (NNMF) [22] and matrix completion using latent Dirichlet allocation (LDA) [3]. All three approaches estimate unknown matrix entries via a low-rank matrix approximation. SVD is based on a singular value decomposition and NNMF is a decomposition under the constraints that all terms in the factoring matrices are non-negative. LDA is based on a Bayesian treatment of a generative topic model. After matrix completion of the *zero* entries in the data matrix, the entries are interpreted as certainty values that the corresponding triples are true. After training, the models can be applied to statistical units in the population outside the sample.

4.2.2 ML-based Transform Plug-ins

Based on the workflow of the SUNS approach described above (see details in [34]) we have designed and developed two *Transform* plug-ins: *RDF2Matrix* transformer and *ProbabilisticRDF* transformer. Both are of type of *InformationSetTransformer* and as defined in the LarKC data layer API, they require *SetOfStatement* as input and produce again transformed *SetOfStatement* as outcome.

[1]Although the completion is applied to the complete matrix, only *zeros* —representing triples with unknown truth values— are overwritten.

Their components can be re-used as or within other types of LarKC plug-ins (Section 4.2.3). The latter is based on the former. Concretely speaking, the core step of the *ProbabilisticRDF* transform, i.e. model learning, needs input data in matrix form which is produced by the *RDF2Matrix* transformer.

RDF2Matrix Transformer

Given predefined statistical unit type and population, the plug-in constructs data matrix by transforming a set of RDF-triples related to statistical units to data matrix. The rows in the matrix stand for instances of a statistic unit and columns represent their features derived from the associated RDF graph. The binary entries *one* and *zero* reflex the truth values of the corresponding triples true and unknown respectively. For example, we have a data set around genetic-variation-relationship between genomes and diseases. Suppose that rows are genomes and columns are diseases. An *one* of the (i,j) entry in the matrix indicates that the i-th gene varies the j-th disease; otherwise it is unknown whether or not a relationship exists between that gene and that disease.

Figure 4.2 shows the architecture of *RDF2Matrix* transformer. Here we see three components and a configuration. To facilitate the usability of the plugin a default configuration will be available in an (TXT or XML) property file. However, user who want to refine the setting of each component has also possibility to set up the configuration by herself. Now let us look at the components one by one. First, the Population Creation component interprets the definition of the statistical unit and population into SPARQL query and executes the query through the LarKC data layer to collect all statistical units, i.e. URIs of key entities. Second, the Sampling component follows some sampling strategy to obtain a representative subset of the population for model training. Finally, the Transformation component extracts input and output features based on the sample, removes those features without significant statistical relevance and afterwards fulfils matrix entries. A great advantage of the SUNS approach is that building matrix is directly based on data and does not necessarily require ontologies.

ProbabilisticRDF Transformer

The plug-in estimates probabilities of the truth value of triples not present yet in the triple store and persistently saves those probabilistic triple in some manner, ideally in quads, thereby they can be then retrieved by (extended) SPARQL queries. The plug-in can be viewed as transform of a conventional triple store into a probabilistic triple store.

Figure 4.3 shows the architecture of *ProbabilisticRDF* transformer plug-in. We can easily see that the first step is to invoke the *RDF2Matrix* transformer converting RDF triple stores to a data matrix. In the next step the Learning component chooses proper algorithm from intern or extern machine learning libraries and trains models. As shown in the Figure 4.3 there is a kind of connector communicating with external (remote) machine learning libraries if necessary. After having learnt models, we estimate the truth value of triples of interest. If desired (in the offline setting), we could add probabilities to original RDF graphs via Probability storing component, so that we create probabilistic RDF graphs.

4.2.3 Reuse of Components

So far we have introduced the basic idea and workflow of the machine learning approach SUNS and provided an overview of the architecture and main components of two *Transform* plug-ins. Now we discuss the reusability of those components.

Population Creation as Identify Plug-in

The definition of the population and the statistical unit could be considered as a identify step in which a relevant sub-graph of a RDF-graph is extracted containing key entities under consideration.

Sampling as Select Plug-in

There are many strategies that can be chosen to sample a representative subset from the population for training with a rational size. The choice depends on the nature of the data and the target relation to predict.

A simple but mostly used strategy is random sampling. In this case the whole population is randomly accessible and all instances can be queried directly from triple stores. Statistical units in the sample for training are randomly sampled and statements predicted for testing are either other randomly selected statistical units (inductive setting) or statements concerning the statistical units in the training sample (transductive setting).

Since the most triple stores are published on the Web, another sampling strategy is link following, i.e. crawling. Assumes that the Web address of one user (i.e., statistical unit) is known and only a subpart of the data can be collected by crawling, which is a typical situation on the Web. Starting from this user profile, the profiles of users connected by certain relationship are gathered by crawling breadth-first and are then added to the training set. The test set is either the same as the training set (transductive setting) or gathered by continued crawling (inductive setting). In this situation, all profiles are (not necessarily directly) connected and generalization of trained models can be expected to be easier since local properties are more consistent than global ones.

Learning as Reason Plug-in

What the Learning component of the *ProbabilisticRDF* transformer dose is nothing else than inductive inference by using machine learning. It materializes then (selected) learned probabilistic triples in LarKC data layer in offline setting or just makes estimation of the true value of triples available for online querying.

5 Reason Plug-in Type

5.1 Introduction

The purpose of a *Reason* plug-in in LarKC is to solve (answer) a user's request (given in the form of a query) by reasoning over the ontological knowledge prescribed by the ontology passed to the reasoner. More concretely, in LarKC the user's request is captured by a SPARQL query which is to be executed by a reasoner against a set of RDF statements passed as parameters. The result of executing such query depends on the type of reasoning operation that is invoked and ranges from a boolean value to a set of variable bindings and includes a set of statements. In addition to the user's query and the data to reason over, a Reason plug-in accepts a contract parameter that defines the behaviour of the reasoner and a context parameter that enables the communication of the plug-in's state among other plug-ins. For further technical details of the Reason interface the reader is referred to Deliverable D5.3.2 - Overall LarKC Architecture and Design [11].

This section is concerned with identifying the key components that constitute a reasoner and different possibilities of reusing such components in other plug-ins or, reusing other plug-ins and components of the LarKC platform for the design of Reason plug-ins. In particular, Section 5.2 describes the design of a rule-based reasoner. Section 5.3 introduces a framework for studying anytime reasoning by ontology approximation systematically through the specification of three independent components. Such framework is specified by a 3-step workflow consisting of a number of independent modules, which can be instantiated for different reasoning tasks and approximation strategies. The crucial elements of this workflow are separate modules for *approximation*, *reasoning* and *evaluation*. After the introduction of this framework we present three concrete ideas on how this 3-step workflow can be realized in the LarKC platform. This includes the description of a single Reason plug-in that implements parts of the aforementioned framework and the description of two LarKC workflows for anytime reasoning by ontology approximation. In every case we highlight the main components and discuss different possibilities of reuse. Section 5.4 describes the design of different variants of a LarKC Reason plug-in that is capable of reasoning with inconsistent ontologies through the use of the PION reasoner, a system for interleaving reasoning and query-based selection for reasoning with inconsistent ontologies.

5.2 Rule-based reasoner

Rule-based reasoning involves the application of rules to infer knowledge. Ullman [37] defines the set predicates whose relations are stored in the knowledge base to be the extensional database. Whereas the set of predicates whose contents are defined in terms of logical rules form the intensional database. This distinction becomes blurred when logical rules exist that infer new tuples for extensional predicates. Therefore, it is generally clearer to discuss ground statements, that exist explicitly within the knowledge base and inferred statements that can be deduced with the combination of ground statements and the logical rules.

A rule can be considered as a simple *IF ... THEN ...* statement, i.e. if a conjugation of atomic statements in the body of the rule are considered true in the model for the knowledge-base then the rule allows for the corresponding head of the rule can also be considered true.

One rule-based formalism that has been thoroughly analysed is Datalog [36], which was originally developed as a database query and rule language. Datalog is based on a simplified version of the Logic Programming language Prolog. The intention was for a rule-system capable of processing large amounts of data from relational databases. Several relevant complexity results of Datalog in regard to query answering have been derived. Querying a static knowledge base in general has polynomial time complexity, but is exponential otherwise [8].

Datalog has a wide variety of applications and can be used for reasoning with other formalisms, including: Description Logic Programming (DLP) [13], language variants from the WSML family [7] and RDF [21]. Disjunctive Datalog, which allows disjunctions in the head of a rule can be used to reason with an even larger subset of OWL DL [17].

Rule-based reasoning has a particular place in LarKC due to the early decision to design the LarKC platform as a SPARQL endpoint. Many of the languages whose formalisms are based on or have an RDF serialisation can have their semantics expressed using rules. This includes RDF and RDFS [28] and the OWL 2 profile OWL2 RL [25]. Interestingly, work on the langauge ELP [20] (a hybrid of DLP [13], EL++ [2] and several OWL2 profiles) has shown that the semantics of more expressive description logics can be adequately represented using rules. Indeed, ELP captures that semantics of OWL2 EL, QL and RL, using a polynomial time algorithm to translate the knowledge base to Datalog.

Very broadly speaking, there are two categories of approaches to reasoning with rule-based formalisms. The first approach is to compute all of the inferences that can be made using the ground statements and logical rules, which uses the rules in a forward chaining manner and hence is often called 'bottom-up'. The result is a minimal model, which contains all statements that are considered true in any valid model of the knowledge base. This process is often called 'materialisation'. The main advantage of this approach is that once computed, the query answering over the minimal model is extremely fast. All that is required is to make a join over each atomic formula in the conjunctive query. However, a major disadvantage with this approach is that the computation can take a very long time for large knowledge bases and the resulting model might be so large that it will not fit in to available storage systems. Furthermore, the technique does not permit (or at least makes it much more difficult) modifications to the extensional database, i.e. the addition or retraction of ground statements.

The second category of reasoning approaches, often called 'top-down' uses the logical rules in the reverse direction, i.e. using the rules in a backward chaining manner. The intention here is to compute only those inferences that will be used to answer a query. The advantage is that potentially far less computation is required and modifications to the extensional database can be made without having to recompute the entire minimal model.

In reality, a whole range of variations, combinations and optimisations of these two categories of approaches exist, each with their own set of advantages and disadvantages.

5.2.1 Rule-based reasoner LarKC plug-in

Rule-based reasoning in the context of the RDF based languages typically involves a rule-set that is applied directly to the RDF triples. Examples of such rule-sets are the RDF and RDFS entailment rules, OWL2 RL and OWL Horst [33] being the default rule-set of OWLIM [19] [1].

For an example, rule 'cax-eqc1' from the OWL2 RL entailment rule set is defined as follows:

$$T(?c1, owl:equivalentClass, ?c2) \wedge T(?x, rdf:type, ?c1) \rightarrow T(?x, rdf:type, ?c2)$$

In plain language, if two classes are equivalent then any member of one class is a member of the other.

The first prototype rule-based reasoner in LarKC uses the IRIS [18] reasoner for its underlying inference engine. Further designs and implementation will lead to a rule-based reasoner for approximate and possibly parallel reasoning.

The prototype rule-based reasoner uses several mechanisms for applying a rule-set directly to the RDF triples. In the context of LarKC, a reasoner accepts a stream of input triples from a previous plug-in (typically a selection component) along with a SPARQL query. The output of the reasoner is then a standard SPARQL result: variable binding for a 'select query', triples for a 'describe' or 'construct' query and a boolean result for an 'ask' query. Various configuration for the internal of a rule-based reasoner plug-n are imagined, but for now, a description of the prototype plug-in is given.

The reasoner plug-in is instantiated with the SPARQL query that needs to be evaluated and the rule-set to be used for inference. Neither of these two inputs are allowed to change during the execution of the plug-in and its owning workflow. During workflow execution, collections of input triples are delivered to the reasoner plug-in by the LarKC platform and supporting components. The data structures for holding RDF statements defined in the LarKC data layer have been carefully designed such that any class that directly or indirectly holds statements implements a common interface, namely 'SetOfStatements'. This common interface allows the input RDF statements to be processed no matter if they are a simple set, a data set, a labelled set or some other data structure.

During each invocation of the reasoner, a collection of RDF statements is taken from the input queue and passed to the reasoner. The IRIS rule-based reasoner plug-in uses these statements and the initial rue-set to perform forward chaining with query answering to generate the query result. The rules are Datalog compatible and can be applied to a relation of ternary tuples, in this case RDF statements or triples. At each iteration, the input RDF statements, the rule-set and the user query are provided to the IRIS reasoner and a complete reasoning/query-answering step is completed using a new knowledge base. For the longer term, such a process does not scale well and a better system for enabling knowledge base updates is planned (it is not considered necessary to implement any kind of retraction mechanism at this stage).

[1] `http://www.ontotext.com/owlim/index.html`

5.2.2 Reuse of sub-components for reasoning

All reasoning systems require the use of a wide range of algorithms and data-structures. Tuples are stored in relations and relations must be indexed to allow tuples to be quickly found, iterated and accessed. For reasoning with formalisms that have an RDF serialisation, there is conceptually a single relation that holds all of the RDF triples (arity 3 tuples) and the logical rules that capture the semantics of the language operate directly on this single relation.

For example, in simple forward chaining mode, IRIS examine the logical rule set and for each rule it will set up and evaluation scheme and indexing specific to the rule, e.g. in the following rule:

$$p(?x, ?y) : -q(?x, ?w, ?z), r(?w, ?z, ?y).$$

IRIS will set up an evaluator for the natural join that occurs between predicates 'q' and 'r' using the variables 'w' and 'z' and will use the bindings for variables 'x' and 'y' to populate the rule head. For this example, the join can be achieved by iterating through every tuple in the relation for 'p', using the second and third terms as values for 'w' and 'z' and using these values to find matching tuples in 'r', thus requiring a compound index on the first and second terms of the relation for 'r'.

The disadvantage here, as for any rule-system that computes a minimal model using forward chaining, is that the model may be very large and its computation intractable. Indeed, this is expected to be the case for most use-cases of LarKC. However, the design of the LarKC platform allows for this drawback to be mitigated and potentially even turned to an advantage. The important characteristic is 'anytime' behaviour, where algorithms are able to return a partial answer, whose quality depends on the amount of computation time they were allowed. The result of an anytime algorithm is an approximation of the full answer.

Taken together, forward chaining and anytime behaviour allow for a fast initial response to a query, with the ability to provide better quality answers over time. As we have seen already, selection components attempt to speed up inference by selecting a subset from all of the available information and using this subset as input to the inference stage. This has been explored not just for scalability purposes, but also as a means to tackle inconsistency (see D4.3.1 - Strategies and Design for interleaving reasoning and selection of axioms).

As well as doing selection before reasoning, including a selection step as part of reasoning itself will allow a marked increase in performance for anytime, rule-based, approximate reasoning. Consider the rule given in the example above. To evaluate this rule in a forward chaining algorithm requires the processing or the entire set of input data. In the LarKC case, this is expected to number in the tens of billions of statements and even on high performance hardware, this computation could easily require hours to complete. The fundamental algorithm required for rule-based reasoning is the ability to execute joins as described above and joins require indexes in order to efficiently find tuples that match is separate relations. However, if selection components (based on user context, statistics, heuristics, etc) are fine-grained enough, then these could be substituted for or used in conjunction with indexes, which would allow faster traversal of relations with the expectation that the initial results would be more likely to satisfy the user. This in turn would mean that the workflow could be terminated earlier.

Therefore, future research activity in LarKC will focus on a more fine-grained integration of selection and reasoning, which can not be achieved at the moment using plug-in wrappers around existing third party components. Instead, reasoners will need to be thoroughly re-examined and a new LarKC prototype reasoner will use selection components for indexes that will speed up inference and simultaneously improve the quality of the initial output. Such behaviour will lead to earlier termination of workflows and hence earlier user satisfaction.

5.3 Anytime Instance Retrieval by Ontology Approximation

The need for approximation on the Semantic Web raises the challenge to develop algorithms for anytime Semantic Web reasoning, and several attempts have been made to find suitable approximation strategies and study their effects in practice [32, 12, 31, 38]. Until now, this work has been limited in scope, has had a rather ad-hoc character (lacking a general framework for theory and application), and most importantly, results have often been inconclusive and show a need for a more thorough experimental analysis. A systematic evaluation of strategies and heuristics is challenging, and the results until now have been difficult to reproduce and compare.

This section, first introduces a framework for testing approximation algorithms systematically that will make the development of such methods easier, and thus increase their chances of adoption and deployment (see Section 5.3.1). To this end, we design a workflow consisting of a number of independent modules, which can be instantiated for different reasoning tasks and approximation strategies. The crucial elements of this workflow are separate modules for *approximation*, *reasoning* and *evaluation*. The first allows implementing approximation strategies by subset selection (eg. subsets of axioms, or of vocabulary), the second allows to specify a specific reasoning task (eg. instance retrieval, ontology classification, or consistency checking), and the final module allows to implement a suitable evaluation metric. This 3-step workflow has been realized into a workbench for studying anytime instance retrieval by ontology approximation. Both, the workbench and the results of our experiments are publicly available online [2].

After introducing the 3-step workflow for studying approximate reasoning we present and discuss three concrete ideas on how this 3-step workflow can be realized in the LarKC platform. First, section 5.3.2 introduces the overall design of a LarKC Reason plug-in that implements part of the 3-step workflow. Such plug-in is described in terms of its constituent components and their functionality. Then, in Section 5.3.3 we present and discuss two alternative LarKC workflows that combine several plug-in types in order to realize our framework for anytime approximate reasoning in LarKC. The purpose of this analysis is to identify the main components that constitute an approximate reasoner and to identify points of reuse in the design of LarKC plug-ins and workflows.

[2]`http://www.few.vu.nl/~gtagni/aboxreasoning/`

5.3.1 A Framework for Anytime Reasoning by Ontology Approximation

Our framework consists of a pipeline of three steps (see Figure 5.1), resulting in a new type of gain diagrams. These three steps allow to define (i) the particular approximation heuristic to be used, (ii) the reasoning task to which it should be applied, and (iii) the definition of a performance measure for evaluating the heuristic. This section describes each of these steps and the resulting gain diagrams.

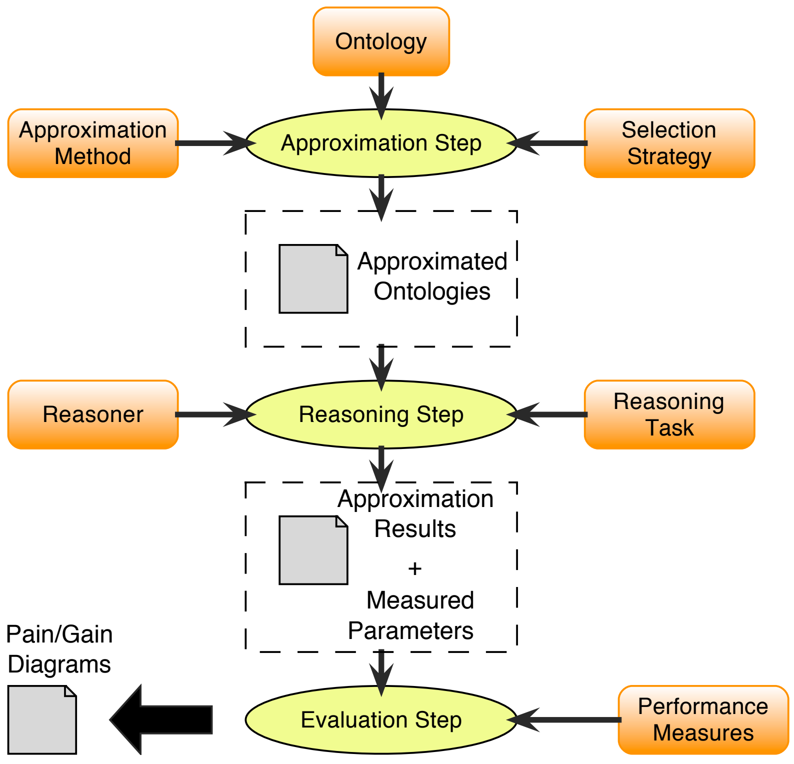

Figure 5.1: 3-step workflow for approximate reasoning experiments; every square box can be changed per experiment

Approximation step

The foundational results from [30] show that performing an approximate reasoning tasks on a logical theory can be transformed into executing a classical reasoner on a suitably approximated theory. Hence, the purpose of the approximation step is to take an ontology O and to return a sequence of approximations $O_1, O_2, \ldots, O_n$. Very often, such approximations can be phrased in terms of a selection method, operating on either the symbols appearing in an ontology (vocabulary selection), or on the set of axioms in an ontology (axiom selection), operating on either or both of the A-box and T-box of the ontology. The approximation component does not depend on a particular strategy: the only requirement is that for a given ontology, this module returns a sequence of approximations. Although our framework imposes no further constraints on the approximation step except that it produces ontologies that can be used in the reasoning step, some formal properties of such selection steps are desirable. Let O^* denote the semantic closure of an ontology, ie. all facts that can be derived according to its semantics. We then have *soundness* if each $O_i^* \subseteq O^*$, ensuring that the approximate results are correct (although possibly incomplete); *monotonicity* if $O_i^* \subseteq O_{i+1}^*$ for all $i = 1, \ldots, n-1$, ensuring that the successive approximations get more correct; and *completeness* if $O^* = O_n^*$, at which point the approximation has reached perfect quality.

Reasoning step

Approximation can be applied to different reasoning problems such as Instance Retrieval or Classification. All that our framework requires is that the reasoning step takes as input an ontology, and returns *answer-sets*. These answer sets could be instance-class memberships (for instance retrieval) or class-class subsumptions (for classification). It is these answer-sets that determine the quality of the approximation, and the computational efforts which determine the price one has to pay.

Analysis step

In the analysis step of our framework we specify the notions of success and costs. These performance measures can be eg. the standard notions of recall (the number of retrieved facts in relation to all possible findings), or precision (the correctness of the given answers), or some more non-standard notion of semantic proximity of the approximate answers to the perfect answers. More generally, we propose a notion of *gain*, which abstracts over the detailed measures and describes the results as ratios between possible and actual findings. *Pain* is the orthogonal notion describing the ratio between the costs of reasoning over an approximate ontology versus the non-approximate one. For specific examples of pain one could think of costs in terms of runtime or other computational resources, such as memory, user-interaction, database access, etc.

Gain-Pain diagrams: Obviously, we are interested in whether the gain (success-ratio of current answers against perfect answers) outweighs the pain (cost-ratio of current answers against perfect answers), in other words in the gain-pain difference. This ratio is plotted in our gain-pain diagrams which show at which point of the anytime computation the gain outweighs the pain (or not, as the case may be, and by how much). Figure 5.2 illustrates these measures. As the quality of the approximation increases along the x-axis from $0 - 100\%$, in this example the gain increases linearly while the pain increases much more slowly initially, and rises more sharply in the final 20%. The combined performance measure (pain-gain curve) is calculated as the difference between these two, with the best performance achieved at about 75% of the approximation where the proportional gain maximally outweighs the proportional pain.

The ideal gain-pain curve rises sharply for the initial approximations of the input representing the desired outcome of a high gain and low pain in the early stages of the algorithm. Although such a convex gain-curve is the most ideal, even a flat gain curve at $y = 0$ is already attractive, because it indicates that the gains grow proportionally with costs, giving still an attractive anytime behaviour.

Notice that gain-curves always start in $(0, 0)$, since for the empty input both gain (e.g. recall) and pain (e.g. runtime) are 0, hence their difference is 0. Gain

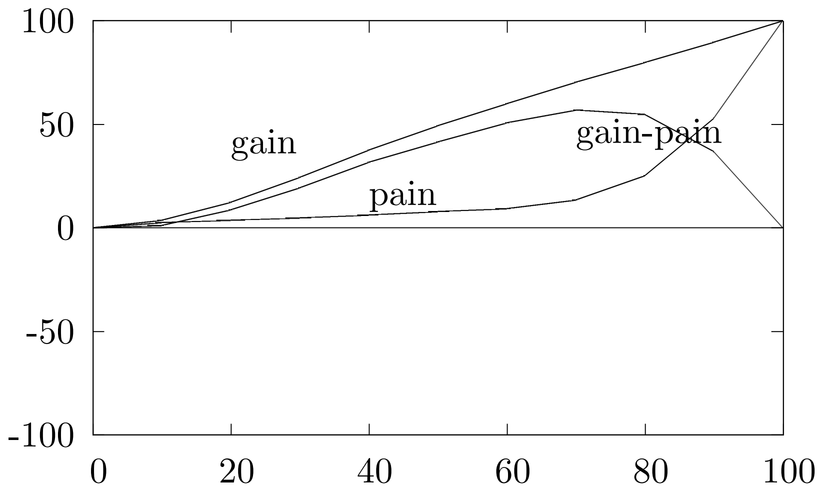

Figure 5.2: gain, pain and gain-pain curves

curves always ends in $(100, 0)$, since for the final perfect approximation both recall and runtime are 100%, hence their difference is again 0. Also notice that gain-pain curves can be negative when the proportional pain outweighs the proportional gain for certain approximations.

5.3.2 Approximate Reasoning Plug-in

The simplest way to implement the 3-step workflow for approximate reasoning in LarKC is as a single LarKC reason plug-in. The modular design of this plug-in allows for changing some of its components and replacing them with other implementations of the same component. For example, the approximation module is an interface that can be instantiated by different classes allowing us in this case to experiment with multiple approximation methods. The same holds for the selection module which allows us to plug different components implementing different selection strategies.

In the following we will discuss the overall design of a LarKC plug-in that implements an anytime, approximate reasoner that implements part of the functionality provided by the workbench described above. Figure 5.3 depicts the overall architecture of our anytime approximate reasoner plug-in illustrating the its components.

Approximation Component This module implements the approximation step of the 3-step workflow for anytime approximate reasoning presented above. Given an ontology this modules returns a set of approximated ontologies. More specifically, this modules defines an interface for approximating ontologies. The interface can be instantiated by different approximation classes each of which implements a specific approximation method. In the current implementation the approximation component takes as input an OWL ontology and returns a sequence (possibly a singleton) of approximated ontologies by selecting a subset of the ontology's vocabulary (atomic concept names) and rewriting the set of terminological axioms according to the approximation method described in [30]. The approximation

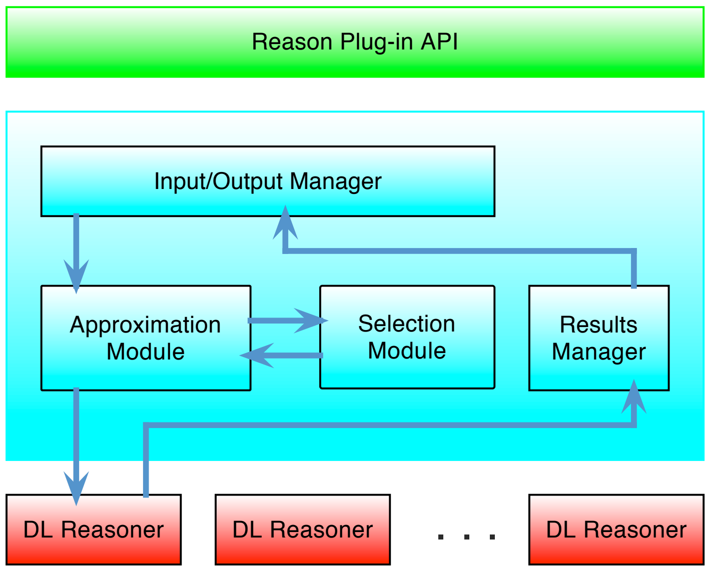

Figure 5.3: Architecture of a reason plug-in in LarKC implementing the 3-step workflow introduced above

module also provides a way to produce incremental approximations of an ontology whereby given an initial ontology it returns a sequence of approximated ontologies (T-boxes) each of them based on an incremental subset of the vocabulary of the ontology. For example, given ontology $O = (T, A)$ the module is able to produce 10 different approximations (T_i, A) where each T_i is an approximated T-box based on 10% of the vocabulary of the ontology. The current implementation of this module approximates only the terminological part of an ontology leaving the assertional part intact. However, it is possible to replace this component by one that approximates both the terminological and the assertional parts of an ontology.

Selection Component The selection component of this plug-in refers to the selection step in the 3-step workflow presented above. This component is implemented as an interface that defines the basic functionality that must be provided by every selection strategy. For the purposes of approximate reasoning we have defined two additional sub interfaces. The first one, a *vocabulary selection strategy* interface, defines the common functionality provided by methods that return a subset of the vocabulary of the ontology. The second one, an *axiom selection strategy* interface, specifies the minimal functionality that must be provided by methods that return a subset of the set of terminological axioms defined in the ontology. A selection module takes an ontology as input and returns either a subset of the vocabulary of the ontology (vocabulary selection strategies) or a subset of the terminological axioms defined in the ontology (axiom selection strategies).

In the current implementation of our workbench we have implemented six vocabulary selection strategies which in the context of this plug-in these selection strategies are to be implemented as six different selection modules. In the following we describe briefly each of the selection strategies.

- *Random (R):* This function randomly selects a set of atomic concept names from the ontology's vocabulary set.

- *Most Referenced (MR):* This function selects concept names according to the number of times they are appear in terminological axioms.
- *Most Members (MM):* This function selects atomic concept names based on the number of instances they have. At each approximation step concepts are sorted according to the number of instances that were retrieved in the previous step. In case there is no feedback from the previous reasoning step concepts are sorted according to the Most Referenced strategy. The rationale behind this strategy is to select as early as possible those concepts that can produce the largest number of instances, thus producing the greatest increase in recall. The main disadvantage of this strategy is that concepts must be sorted at each step of the approximation process. In addition to this, the strategy must be combined with another strategy to produce an initial ordering.
- *Restriction Class (RC):* This function gives higher priority to the fillers of quantified concept expressions and to their respective sub concepts. If the number of such elements is less than desired number M the additional concepts are chosen based on the number of instances asserted in the assertional part of the ontology. The rationale of this strategy is that property restrictions are used for defining classes implicitly. Consequently, these classes may contribute to retrieving a large number of instances. The main disadvatange of this strategy is that not every class in an ontology is defined through property restrictions, a characteristic that makes this strategy incomplete. Therefore, as with the previous strategy this one needs to be complemented with another strategy for selecting classes that are not defined through property restrictions.
- *Most Direct Subclasses (MDS):* This function selects atomic concepts based on the number of direct subclasses they have. The first time this strategy is used, atomic concepts are sorted in decreasing number of direct subclasses and each successive call to this function returns the next set of concepts. As with the Most Referenced strategy concepts can be sorted only once at the beginning of the anytime reasoning process.
- *Least Direct Subclasses (LDS):* This function is the opposite of the MDS function. The rationale for this strategy is that concepts with the least number of subclasses are more specific and tend to be used to annotate large number of individuals.

Input Manager This component is responsible for splitting the ontology into its terminological and assertional parts. This is required since the current implementation approximates only the terminological part of an ontology. An advantage of separating the terminological from the assertional part is that this allows the approximate reasoner to combine a single terminology with multiple pieces of instance data.

Reasoner Although the reasoner component could be built-in into the approximate reasoner plug-in we have decided to leave it outside the reasoner. The main advantage of doing so is that it allows us to (re)use multiple standard DL reasoners. In the current design access to DL reasoners is accomplished through either the OWLAPI or the OWLLink interfaces.

Using the context parameter of the Reason interface it is possible to invoke the approximate reasoner and request to answer the same query using different approximations of the same ontology. A decider, or the end-user application, would invoke the reasoner providing a SPARQL query and a data set. The reasoner would then solve the query by using an approximated ontology as computed by the approximation module and return the results back to a LarKC Decider or user application. In case further reasoning is required the approximate reasoner could be invoked again with the same query and the same input ontology. This time, the reasoner would approximate the ontology using a bigger subset of the vocabulary and the solve the query using this new approximated version of the ontology. The context parameter in the Reason interface could be used to keep state-related information between calls to the reasoner, for example, to control the size of the subset of the vocabulary upon which the next approximation should be computed.

5.3.3 Reuse of sub-components for Anytime Approximate Reasoning

The previous section described the overall architecture of a LarKC Reason plug-in that implements part of the functionality provided by the 3-step workflow introduced above in terms of a series of modular components for *selection*, *approximation* and *reasoning*. Although the plug-in's architecture is rather simple, with only three main (sub)components, it highlights several points of reuse. In the following we will identify these points and discuss how to implement the functionality provided by the approximate reasoning plug-in as a LarKC workflow using several plug-in types.

- *Approximation as Transformation plug-in:* An *InformationSetTransformer* plug-in takes as input an *InformationSet*, eg. the URL of an OWL ontology, and returns a (transformed) version of the input *InformationSet*. One possible implementation of such interface could return an approximated version of the input ontology, i.e. the transformation step consists in approximating the ontology. Note that this is literally the case of our approximate reasoner as the approximation of an ontology is defined in terms of rewriting the terminological axioms of the ontology based on a subset of the vocabulary. Such a transformer plug-in would only have to invoke the specific selection method and allow for context information to be passed to it in order to implement incremental subsetting of the vocabulary.

- *Approximation as Selecter plug-in:* Another possibility is to implement the approximation module as an instance of a *Selecter* plug-in interface. In this case, the input parameter of type *SetOfStatements* represents the ontology that needs to be approximated. The plug-in's output (of type *SetOfStatements*) represents the approximated version of the input ontology. As with the Transformation-based approximation discussed above, a selection plug-in would have to invoke a specific selection method to select the subset of the ontology's vocabulary or, alternatively, each selection strategy could be implemented as a different selection plug-in that not only implements a specific selection function but also approximates an ontology based on this selection method. The selection plug-in should also be able to accept context information that allows the client class to pass state-related information, eg. the percentage of vocabulary that must be selected in the current call.

- *Selection as Selecter plug-in:* In case the approximation of an ontology is based on selecting a subset of the terminological axioms of the ontology the selection module's functionality could be implemented by a *Selecter* plug-in. Such plug-in would return a *SetOfStatements* representing a subset of the terminological axioms. Different axiom selection strategies could be implemented by different instances of this *Selecter* interface.

- *Reasoning Component as Reason plug-in:* An obvious point of reuse in our approximate reasoning plug-in is the use of a reasoner component. The functionality of such module could easily be implemented by a separate LarKC plug-in that provides alternative reasoning capabilities in terms of expressivity, computational resources, reasoning paradigm, etc. In particular, for the implementation of our reasoner we are planning to use standard DL reasoners and access them through the OWL API and OWLLink plug-ins that provide a wrapper component over existing OWL reasoners.

Two Workflows for Anytime Approximate Reasoning

Figure 5.4 depicts a LarKC workflow for studying anytime reasoning by ontology approximation. The workflow consists of four plug-in types. The *Identify* plug-in is responsible for identifying the data over which reasoning will be done to answer a given query. The *Selecter* plug-in receives the ontology identified by the previous plug-in and returns an approximation of that ontology. Depending on the desired selection strategy and approximation method different instances of this plug-in will be invoked and executed. Once the *Selecter* plug-in computes the approximated version of an ontology an appropriate *Reason* plug-in can be invoked with the approximated ontology and original query as parameters. Here, the selection of the reasoner may depend on many factors such as QoS parameters. The *Reason* plug-in is responsible for answering the given query using the given approximated ontology. The specific instance of this plug-in type could be one that invokes an external DL reasoner through the OWL API paradigm or OWLLink protocol. The last component of the workflow is a emphDecider plug-in. The *Decider* is responsible for controlling the execution of the workflow and deciding whether further approximation must be done in order to produce more accurate results. For this, the decider must be capable of keeping track of the state between multiple invocations to the selection plug-in. This information is part of the context information that is passed to the *Selecter* plug-in in each invocation and could include the percentage of vocabulary to be chosen in the current approximation step.

One way the Decider could determine whether to use another approximated version of the ontology could be by analyzing the data obtained from the Gain-Pain diagrams produced by the *Evaluation* step of the 3-step workflow for approximate reasoning, assuming the *Decider* (or any other component in the LarKC workflow) implements the metrics for evaluating the results returned by the reasoner.

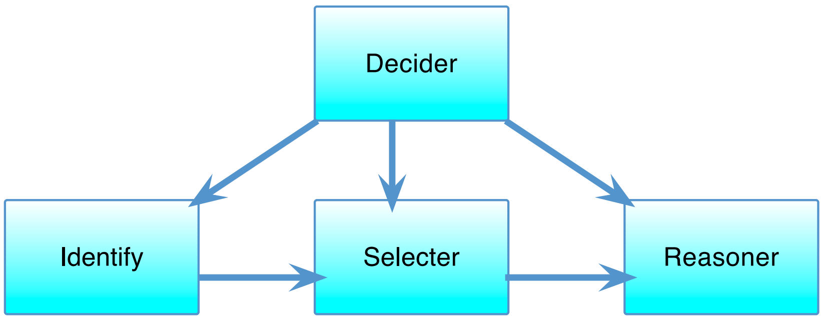

Figure 5.4: LarKC workflow for anytime reasoning by ontology approximation using a Selecter to approximate ontologies

Figure 5.5 illustrates the overall design of a second LarKC workflow where the *Selecter* plug-in has been replaced by a *InformationSetTransformer* plug-in that is responsible for approximating a given ontology. As in the previous case, different approximation methods and selection strategies could be implemented by different *InformationSetTransformer* plug-ins.

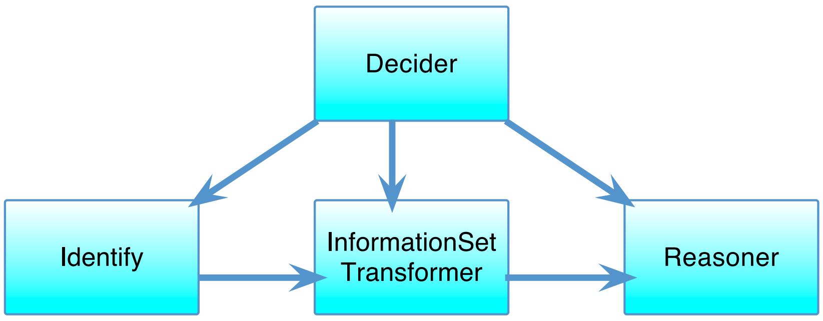

Figure 5.5: LarKC workflow for anytime reasoning by ontology approximation using a Transformer to approximate ontologies

5.4 Reasoning with Inconsistencies in PION

Re-using and combining multiple ontologies on the Web is bound to lead to inconsistencies between the combined vocabularies. Even many of the ontologies that are in use today turn out to be inconsistent once some of their implicit knowledge is made explicit. The classical entailment in logics is explosive: any formula is a logical consequence of a contradiction. Therefore, conclusions drawn from an inconsistent ontology by classical inference may be completely meaningless. That appeals for a system which can reasoning with inconsistent ontologies and return meaningful answers.

In [14], we develop a general framework of reasoning with inconsistent ontologies, in which relevance based selection functions are used to obtain meaningful answers, where the meaningfulness is interpreted as the answer is supported by a selected consistent sub-ontology of the inconsistent ontology, and its negative answer is not supported. The main idea of the framework is: given a selection function, which can be defined on the syntactic or semantic relevance, we select some consistent sub-theory from an inconsistent ontology. Then we apply standard reasoning on the selected sub-theory to find meaningful answers. If a satisfying answer cannot be found, the relevance degree of the selection function is made less restrictive thereby extending the consistent sub-theory for further reasoning. Namely, a system for reasoning with inconsistent ontologies is designed to be one which make a processing of interleaving reasoning and selection.

PION is a system of interleaving reasoning and query-based selection for reasoning with inconsistent ontologies[3]. In PION, selection functions play the main role for query-based selection. The selection function can either be based on a syntactic approach, like Chopra, Parikh, and Wassermann's syntactic relevance [6] and those in PION[14], or based on semantic relevance like for example in computational linguistics as in Wordnet [4] or based on semantic relevance which is measure by the co-occurrence of concepts in search engines like Google[15].

In PION, selection functions are designed to query-specific, which is different from the traditional approach in belief revision and non-monotoic reasoning, which assumes that there exists a general preference ordering on formulas for selection. Given a knowledge base Σ and a query ϕ, a selection function s is one which returns a subset of Σ at the step $k > 0$. Let $\mathbf{L}$ be the ontology language, which is denoted as a formula set. A selection function s is a mapping $s : \mathcal{P}(\mathbf{L}) \times \mathbf{L} \times N \rightarrow \mathcal{P}(\mathbf{L})$ such that $s(\Sigma, \phi, k) \subseteq \Sigma$.

A selection function s is called *monotonic* if the subsets it selects monotonically increase or decrease, i.e., $s(\Sigma, \phi, k) \subseteq s(\Sigma, \phi, k + 1)$, or vice versa. For monotonically increasing selection functions, the initial set is either an emptyset, i.e., $s(\Sigma, \phi, 0) = \emptyset$, or a fixed set Σ_0. For monotonically decreasing selection functions, usually the initial set $s(\Sigma, \phi, 0) = \Sigma$. The decreasing selection functions will reduce some formulas from the inconsistent set step by step until they find a maximally consistent set.

Traditional reasoning methods cannot be used to handle knowledge bases with large scale. Hence, selecting and reasoning on subsets of Σ may be appropriate as an approximation approach with monotonically increasing selection functions. Web scale reasoning on a knowledge base Σ can use different selection strategies to achieve this goal. Generally, they all follow an iterative procedure which consists of the following processing loop, based on the selection-reasoning-decision loop discussed above:

1. select part of the knowledge base, i.e., find a subset Σ'_i of Σ where i is a positve integer, i.e., $i \in I^+$;

2. apply the standard reasoning to check if $\Sigma'_i \models \phi$;

3. decide whether or not to stop the reasoning procedure or continue the reasoning with gradually increased selected subgraph of the knowledge graph (Hence, $\Sigma'_1 \subseteq \Sigma'_2 \subseteq ... \subseteq \Sigma$).

Monotonically increasing selection functions have the advantage that they do not have to return *all* subsets for consideration at the same time. If a query can

[3]http://wasp.cs.vu.nl/sekt/pion

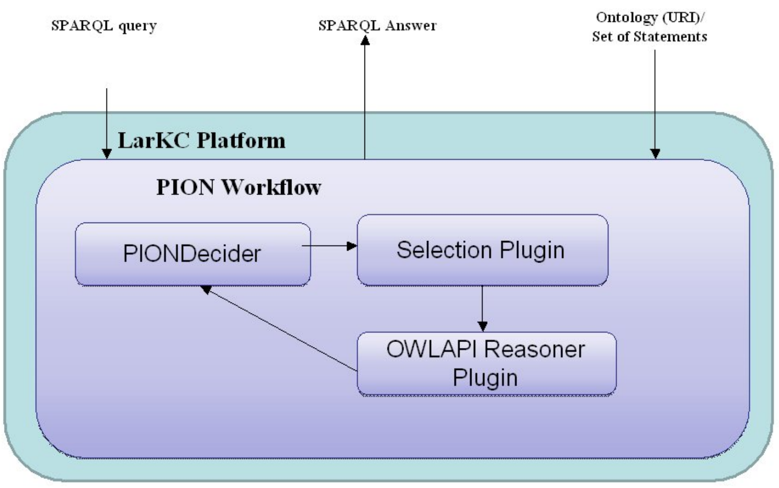

Figure 5.6: Architecture of the PION Reason Plug-in

be answered after considering some consistent subset of the knowledge graph KG for some value of k, then other subsets (for higher values of k) don't have to be considered any more, because they will not change the answer of the reasoner.

The general scenario of interleaving reasoning and selection for PION is shown in Figure 5.6. Namely, the system replies on a decider to manipulate a selecter for selecting some relevant axioms and call a reasoner for standard reasoning over semantic data. A PION system can deal with both consistent ontologies and inconsistent ontologies. At the beginning step of interleaving process, the system first checks whether or not the targeted ontology data are consistent. If the ontology is consistent, then the system would use the reasoner to make the standard reasoning without the non-trivial interleaving processing. If the ontology is inconsistent, then the system will start the interleaving processing for selection and reasoning.

There are various scenarios of interleaving reasoning and selection for PION with the LarKC platform.

- **DIGPION**. DIGPION is one in which an external PION reasoner is called via the DIG interface plug-in inside the LarKC platform. The main advantage of DIGPION is that we can rely on an externally implemented PION system for interleaving reasoning and selection with the LarKC Platform.

- **SimplePION**. SimplePION is one in which PION is implemented as a plug-in with some simplified functions, which include the support of standard boolean answers (i.e., either "true" or " false) without using the three-valued answers such as "accepted", "rejected", and "undertermined", which has been proprosed in [14].

- **PIONwithStopRules**. PIONwithStopRules is one in which PION uses some stop rules to decide when it would stop the selection and jump to provide its reasoning result. The idea of using stop rules is inspired by the investigation of huamn and animal search strategies in Biology and Cognitive Science. As discussed in LarKC deliverable D4.2.2[23], knowing when to stop is one of the most fundamental problems when engaging in any type of activity. Most real-world problems do not have a pre-defined completion

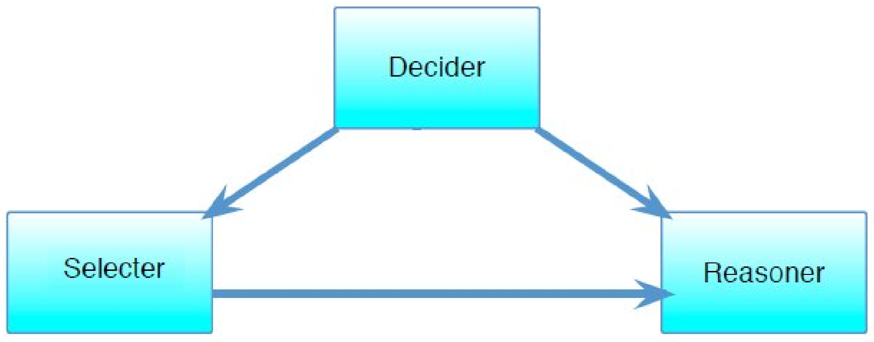

Figure 5.7: PIONWorkflow

criterion. The problem of search termination resurfaces in an aggravated form when a system faces more than a single problem at once. When time and effort need to be allocated to multiple tasks finding the right moments to switch between tasks constitutes a difficult optimization problem. The interleaving framework can be considered as a processing which switches the tasks of reasoning and selection.

- **PIONWorkflow**. PIONWorkFlow is one in which PION is designed as a workflow which use selection plug-ins and reasoner plug-ins, as shown in Figure 5.7. At the beginning of the PION workflow processing, the decider first checks if the the ontology is consistent. If the ontology is consistent, then the decider will start the standard reasoning processing, namely, use the standard OWLAPI reasoner to obtain the result. If the ontology is inconsistent, then the decider will start a non-standard reasoning processing, namely an interleaving processing. The main advantage of the scenarios of PIONWorkflow is that this approach provides the possibility to use various selecters and reasoners which have been implemented independently from the interleaving framework.

All of those variant plug-in/workflows of PION are designed for the reusability with the LarKC platform. They can be used to serve as a reasoner which can deal with both consistent and inconsistent ontologies, although using a PION reasoner plug-in/workflow for reasoning over consistent ontology would lead to more overheading in the interleaving processing. The DIGPION scenario can be used to gain the full functionality support from the existing popular ontology reasoners such as RacerPro, FACT++, and KAON2. The SimplePION plug-in can be used to gain the basic support from the build-in OWLAPI reasoner for reasoning over OWL data. The PIONwithStopRules plug-in is expected to be used for dealing with large scale data with some loss of the expected answers. The PIONWorkFlow provides the possibility to use different selection plug-ins and reasoner plug-ins for the interleaving processing.

6 Design Patterns

6.1 Introduction

A design pattern is a general reusable solution to a commonly occurring problem [1]. According to [9], to be useful, a pattern should clearly specify the problem(s) which it can solve and the context in which such problems arise and in which the solution is recommended. In addition, it should contain best practices and guidelines to document the most important aspects which ensure a high quality of the artefact (or system) to be built. A good pattern must achieve the optimal balance between generality and applicability; the idea expressed in a pattern should be general enough to be applied in various settings, but still specific enough to provide effective support.

Patterns are important in LarKC at the conceptual and engineering levels. The engineering of the LarKC platform can make use of various types of patterns at analysis, design, implementation and testing time.

In the scope of WP5 The Collider Platform, different design patterns have been identified as a result of the analysis of performance and scalability of the LarKC platform architecture and current prototype implementations. These design patterns are modelling the features of different kind of LarKC workflows, which may present some limitations on performance and scalability, under certain deployment conditions, and specifying concrete requirements with regards to performance and scalability improvement. The patterns are proposing solutions to the identified problems through the application of different techniques in the implementation, deployment and execution of the workflows (such as parallelization, distribution and remote execution, data partitioning, etc.). The definition of each pattern is illustrated with one or more examples of concrete LarKC use cases. Each patterns proposed solution will therefore be applicable to similar situations than the given examples.

According to [9], a pattern describes a proven and tested solution to a common problem within a specific context. However, it is possible to describe yet non-tested ideas in the form of a pattern. The owner of the term pattern language, the architect Christopher Alexander [5], associates a rating to each pattern to indicate how well they are proven in real-world examples. It must be noted that the definition of the design patterns described in this section is still in a preliminary stage and its application to real-world examples (in our case, LarKC use cases) is in progress. The design patterns descriptions will evolve as the tests are progressing on concrete LarKC use cases. It is also possible that new patterns are identified. Updates will be reported in future WP1 and WP5 deliverables.

The description of the design patterns within this deliverable is structured as follows:

- Pattern name: meaningful name which identifies the pattern
- Context: description of the situation where the problems and proposed solutions apply
- Problem: description of the problem which the pattern intends to solve

[1] `http://en.wikipedia.org/wiki/Design_pattern_(computer_science)`

- Solution: proposed solution to the above described problem, according to the concrete context conditions
- Examples: real-world examples in which this pattern can be identified and hence, the proposed solution can be applied
- Other related patterns: other patterns that should also be considered as an alternative or complement to solve the given problem

The terms "expensive" and "cheap", refer to computation, are related, in most cases, to execution time measured in relation to the accompanying processes within the same workflow or to the complete workflow execution.

The design patterns related to performance and scalability of LarKC workflows are described in the following subsections.

6.2 Expensive Pre-computation Followed by Cheap Computation

- *Pattern name:* $1 * Expensive + N * Cheap$
- *Context:* Figure 6.2 depicts the structure of this design pattern. The pattern applies when the execution workflow includes
  - One expensive computation process at the beginning of the workflow execution, known as pre-computation (pre-processing).
  - Cheaper computation many times, after the pre-computation.

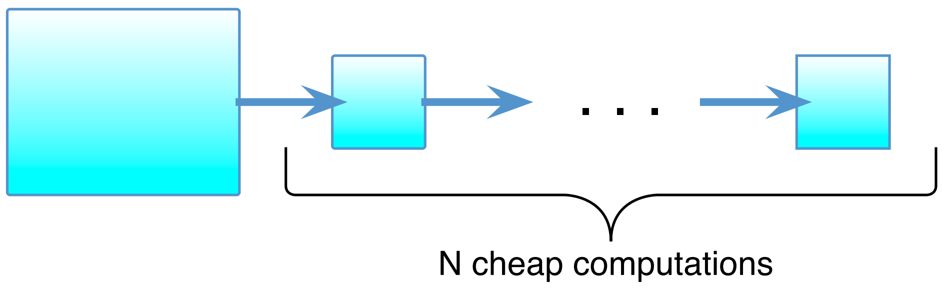

Figure 6.1: Design Pattern 1*Expensive + N*Cheap

- *Problem:* The pre-computation stage, being very expensive in relation to the following stages within the workflow, is penalizing the performance of the overall workflow execution.
- *Solution:*
  - Execution of the pre-computation stage using more powerful hardware resources, remotely located (such as a HPC cluster) and execution of the cheaper steps in the local resources (where the platform is running)
  - Steps:

        * pre-staging (deployment of the pre-computation process in the remote resources)
        * processing (data driven - processing within the remote resources). This may be split in different steps (processing 1, processing 2, etc.), being then either different instances of the same process, running over different data chunks ("trivial" parallelization), or different parts of one single process (parallelization within plug-in)
            · processing 1
            · processing 2
            · processing 3
            · ...
        * post-staging (moving the data back to the local resources)
        * post-processing (query driven real time interaction workflows, composed by cheaper processes)
    - Suggested technology: JavaGAT [2] with predeployment
    - Conditions when this solution is applicable: the overhead (in terms of time) added by the remote execution must be low in comparison to the execution time of the pre-computation

- *Examples:* "real-world" examples (LarKC use cases) in which this pattern can be identified and therefore, the proposed solution can be applied
    - Semantic annotation generation (WP7a)
    - Map-Reduce application to (part of) the WP7a workflow (computing the closure once) and then do querying (several times)
    - WP2 selector (very expensive selection), followed by other (cheap) plug-ins execution within the workflow

- *Other related patterns:* other patterns that should also be considered as an alternative or complement to solve the given problem
    - The pre-computation step may follow one of the following patterns:
        * Design Pattern 4 - Replication + Partitioning: parallelization "across instances of the same plug-in" [3], running different instances of the same process over different data chunks (for this we need a partitioning function, data-dependent).
        * Other: parallelization "within a plug-in" [4]

[2]JavaGAT middleware supports the remote deployment and execution of plug-ins within LarKC, `http://www.cs.vu.nl/ibis/javagat.html`

[3]parallelization "across plug-ins" or "across instances of the same plug-in": loosely coupled components (either different plug-ins or different instances of the same plug-in) that are executed at the same time (in parallel), in order to achieve an improvement in the performance. Communication between the parallel components takes place only at the beginning and at the end of the execution (and possibly during the execution, but not frequently)

[4]Parallelization "within a plug-in": applied to the algorithms that constitute the plug-ins (inside the plug-in)

6.3 Expensive Computation + Cheap Computation + Access to External Resources

- *Name of the Pattern:* $N * Expensive + N * Cheap + External resources$
- *Context:* Figure 6.3 depicts the structure of this design pattern. The pattern applies when the execution workflow includes:
  - Some plug-ins which are more expensive than others
  - They run over different data sets
  - They run in different resources

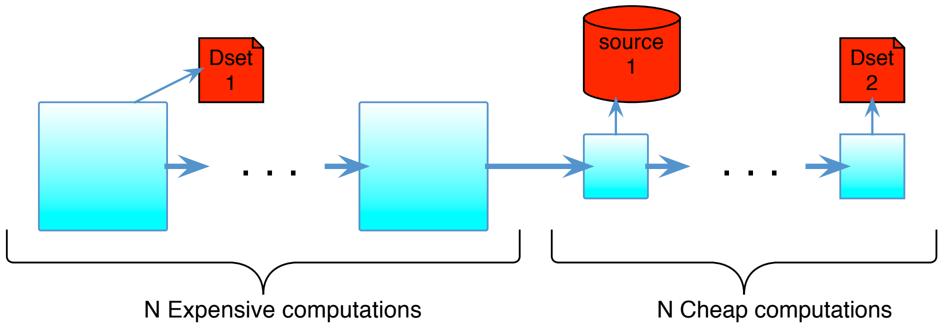

Figure 6.2: Design Pattern 2: N*Expensive + N*Cheap + External resources

- *Problem:* The expensive stages are penalizing the performance of the overall workflow execution. Due to the fact that they may access external data sources, predeployment of the datasets is not feasible.
- *Solution:*
  - Execution of the expensive stages in more powerful hardware resources, remotely located (such as a HPC cluster) and execution of the cheaper steps in the local resources (where the platform is running). An alternative is the execution of the complete workflow in powerful hardware resources which allow access to external data sources.
  - Suggested technology:
    - Alternative 1: JavaGAT with no pre-deployment. For an introduction to JavaGAT the reader is referred to [11]
    - Alternative 2: Tomcat.
  - Conditions when this solution is applicable: the overhead (in terms of time) added by the remote execution must be low in comparison to the execution time of the complete workflow. This applies mainly for JavaGAT technology, which introduces more overhead than Tomcat.
- *Examples:* "real-world" examples (LarKC use cases) in which this pattern can be identified and therefore, the proposed solution can be applied
  - WP6 Alpha Urban LarKC workflow including:

    * identification of external data sources (query driven): cheap;
    * transform data to RDF: expensive. This is a good example to use a powerful machine equipped with multiple CPUs in order to run the transform task in parallel on different input chunks. Id propose here to use the Tomcat approach since its overhead is lower than for JavaGAT.

- *Other Related Patterns:* other patterns that should be also considered as an alternative or complement to solve the given problem
  - The expensive steps may be optimized applying one of the following patterns:
    * Design Pattern 4 - Replication + Partitioning: parallelization "across instances of the same plug-in", running different instances of the same process over different data chunks (for this we need a partitioning function, data-dependent).
    * Other: parallelization "within a plug-in"

6.4 Continuous Expensive Computation

- *Name of the Pattern:* $\infty * Expensive$
- *Context:* Figure 6.4 depicts the structure of this design pattern. The pattern applies when the execution workflow includes:
  - Indefinite iterations of one expensive computation plug-in

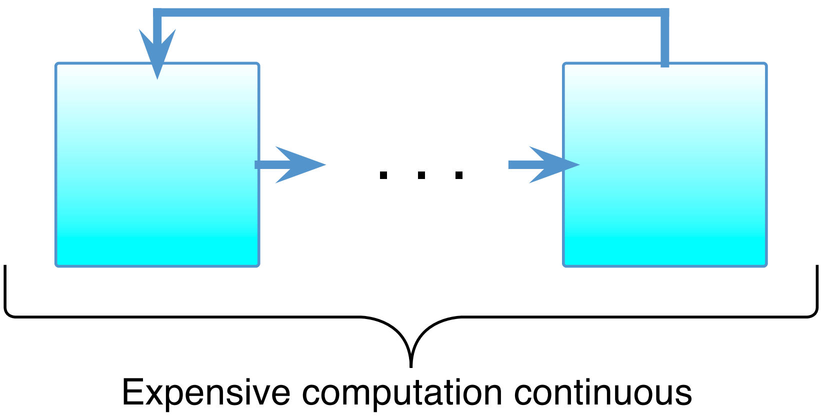

Figure 6.3: Design Pattern 3: $\infty * Expensive$

- *Problem:* The execution of the expensive (unique) plug-in is penalizing the performance of the overall workflow execution.
- *Solution:*
  - Execution of the complete workflow in more powerful hardware resources (such as a HPC cluster), either remotely located or together with the platform itself.

- Suggested technology:
  - JavaGAT with no pre-deployment, in case the workflow is accessing external data sets which may change over the time. For an introduction to JavaGAT the reader is referred to [11]
  - JavaGAT with predeployment, in case it is possible to deploy the data set prior to the workflow execution (that is, in case it is coming from external sources, it is not changing over time).
- Conditions when this solution is applicable: the overhead (in terms of time) added by the remote execution must be low in comparison to the execution time of the complete workflow.

- *Examples:* "real-world" examples (LarKC use cases) in which this pattern can be identified and therefore, the proposed solution can be applied
  - WP7a: Semantic Integration (continuous improvement of a dataset).
  - Computing the closure of a dataset applying MapReduce.
- *Other Related Patterns:* other patterns that should be also considered as an alternative or complement to solve the given problem
  - Some of them or all expensive steps may be optimized applying one of the following patterns:
    - Design Pattern 4 - Replication + Partitioning: parallelization "across instances of the same plug-in", running different instances of the same process over different data chunks (for this we need a partitioning function, data-dependent)
    - Other: parallelization "within a plug-in"

6.5 Replication + Data partitioning

- *Name of the Pattern: Replication + Partitioning*
- *Context:* Figure 6.5 depicts the structure of this design pattern. The pattern applies when the execution workflow includes
  - One plug-in ($P1$) which can be instantiated several times in parallel $(1, \ldots, n)$, over different partitions of a data set
- *Problem:* The execution of the expensive (unique) plug-in is penalizing the performance of the overall workflow execution.
- *Solution:*
  - Execution of the complete workflow in more powerful hardware resources (such as a HPC cluster), either remotely located or together with the platform itself. A partitioning function, data-dependent, is necessary to be executed prior the execution of the plug-in instances.
  - Suggested technology:

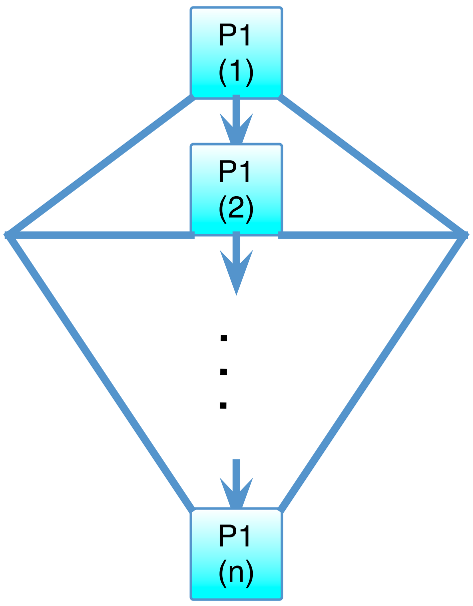

Figure 6.4: Design Pattern 4: Replication + Partitioning

    * Marvin-style implementation [24]
    * Java Thread Pooling
  - Conditions when this solution is applicable:
    * the overhead (in terms of time) added by the remote execution must be low in comparison to the execution time of the complete workflow;
    * it is possible to deploy the complete data set prior to the workflow execution (that is, in case it is coming from external sources, it is not changing over time);
    * input data is splittable.

- *Examples:* "real-world" examples in which this pattern can be identified and therefore, the proposed solution can be applied
  - WP7b: Semantic annotation, information extraction application (GATE).
  - LarKC plug-ins which operate on splittable data sets.
- *Other Related Patterns:* other patterns that should be also considered as an alternative or complement to solve the given problem
  - This design pattern may be applied to optimize expensive steps within other design patterns.

6.6 Balanced Computation

- *Name of the Pattern:* $N * (X * Expensive + Y * Cheap)$
- *Context:* Figure 6.6 depicts the structure of this design pattern. The pattern applies when the execution workflow includes:

- Some expensive steps, combined with some cheaper ones, but no one of them dominating

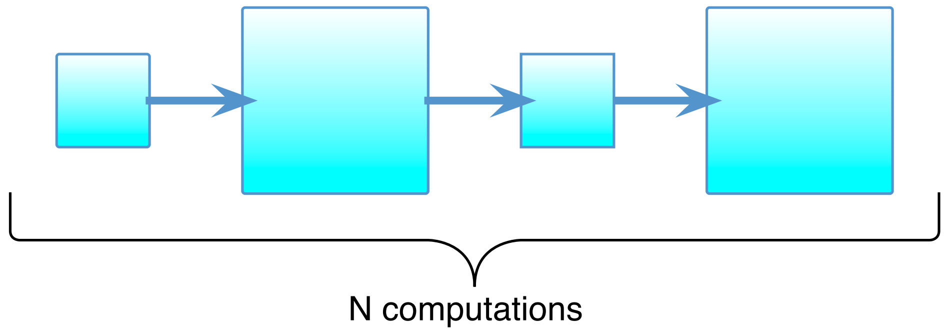

Figure 6.5: Design Pattern 5: $N * (X * Expensive + Y * Cheap)$

- *Problem:* The execution of the expensive plug-ins is penalizing the performance of the overall workflow execution.
- *Solution:*
  - The big plug-ins are shipped to (remote) more powerful hardware resources (a bigger machine, which imposes low overhead, but not a HPC cluster).
  - Suggested technology:
    - Tomcat, WS-style architecture
  - Conditions when this solution is applicable:
    - The overhead of shipping code and data to the remote (more powerful) resources is not too big compared with the execution time of the complete workflow.
- *Examples:* "real-world" examples (LarKC use cases) in which this pattern can be identified and therefore, the proposed solution can be applied
  - WP6 Alpha Urban LarKC workflow
- *Other Related Patterns:* other patterns that should be also considered as an alternative or complement to solve the given problem
  - Some of them or all expensive steps may be optimized applying one of the following patterns:
    - Design Pattern 4 - Replication + Partitioning: parallelization "across instances of the same plug-in", running different instances of the same process over different data chunks (for this we need a partitioning function, data-dependent).
    - Other: parallelization "within a plug-in"

7 Conclusion

This deliverable reported on the work that is being done in LarKC towards the definition of an operational framework for scalable reasoning in LarKC. The main goal of this deliverable (the second in a series of three deliverables concerned with the definition of the aforementioned framework) was to report on the results of a series of activities aimed at improving re-use and tighter integration of components in LarKC. To achieve that goal we have presented the work on several plug-ins currently being implemented in LarKC and described their overall architecture. We have made special emphasis in exploring and identifying different possibilities for re-use. For each plug-in type described in this deliverable we have identified their main components and discussed how they can be implemented as separate plug-ins in order to foster re-use and, how they can re-use other components/plug-ins to implement their functionality.

The second contribution of this deliverable is the result of the close collaboration between technical work packages 1 and 5 and comes in the form of a series of design patterns. These aim at supporting developers in the implementation of plug-ins. These design patterns are the result of the analysis of performance and scalability of the LarKC platform's architecture and current prototype implementations.

From the work on user interest-based selection we were able to identify the need for a platform-wide mechanism for representating the users interest. The suggestionn is to add a series of classes to the LarKC API to represent user interests. In this way, every plug-in would have access to this information without the need of implementing a model of user interests in each plgu-in. Such set of classes should provide the basic functionality for accessing the user's contextual information, like for instance the user's interests.

The work on ML-based data transformation allowed us to identify several possibilities for reuse of components in the platform. In particular, we have identified the possibility to extract from the data transformation plug-in the mechanism for creating populations and implement this functionality as an Identify plug-in. In this way population-based plug-ins (those that work based on populations) can reuse the same Identify plug-in. We have also identified the possibility to use Select plug-ins to implement sampling mechanisms. How to do effective and efficient sampling of data is being investigated in the context of the technical work package 2. The work on ML-based transformation has also highlighted the need for inductive reasoners in LarKC.

The work on Reason plug-ins have shown that the relation between selection and reasoning is an important one that needs to be further investigated and exploited. For some type of reasoning paradigms (e.g. rule-base reasoning) a more ne-grained integration of selection and reasoning need to be investigated; which can not be achieved at the moment using plug-in wrappers around existing third party components. We have also reported the ongoing work on instance retrieval where a framework for anytime instance retrieval by ontology approximation has been presented. We have also presented and discussed three concrete ideas on how the 3-step workow provided by this framework can be realized in the LarKC platform. Moreover, we have discussed several possibilities for reusing some of the components of the approximate reasoning plug-ins introduced in this deliverable, namely the possibility to implement approximate reasoning as a Transform and/or Select plug-ins. Future work will investigate the possibility to re-use existing Select components (developed in technical work package 2) within the approximate reason plug-in. We have also reported on the work on PION, a system for reasoning with inconsistent ontologies, and discussed ideas about how the selection component in PION can be realized by several Select plug-ins.

Future work will be concerned with the specification of the operational framework based on the lessons learned in the first two deliverables and with keep working on fine-grained reuse of components in LarKC.

References

[1] AtanasKiryakov, ZdravkoTashev, Damyan Ognyanoff andRuslanVelkov andVassilMomtchev andBoikoBalev, and IvanPeikov. D5.5.2 - ValidationgoalsandmetricsfortheLarKCplatform, August 2009. Available at http://www.larkc.eu/deliverables/.

[2] F. Baader, S. Brandt, and C. Lutz. Pushing the el envelope further. In *Proceedings of the Washington DC workshop on OWL: Experiences and Directions*, 2008.

[3] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3, 2003.

[4] Alexander Budanitsky and Graeme Hirst. Semantic distance in wordnet: An experimental, application-oriented evaluation of five measures. In *Workshop on WordNet and Other Lexical Resources, 2nd meeting of the North American Chapter of the Association for Computational Linguistics*. Pittsburgh, PA., 2001.

[5] M. Silverstein C. Alexander, S. Ishikawa. *A pattern language: towns, buildings, construction*, volume 2 of *Center for Environmental Structure*. Oxford University Press US, 1977.

[6] Samir Chopra, Rohit Parikh, and Renata Wassermann. Approximate belief revision prelimininary report. *Journal of IGPL*, 2000.

[7] J. de Bruijn, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, M. Kifer, and D. Fensel, year=2005. D16. 1v0. 2 The Web Service Modeling Language WSML.

[8] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions on Database Systems (TODS)*, 22(3):364–418, 1997.

[9] E.Simperl, U. Keller, F. Fischer, B. Bishop E. Oren, Z. Huang, G. Tagni, J. Quesada, B. Fortuna, J. Hu, and Y. Qin. An overview of relevant work in other areas. Technical report, Large Knowledge Collider (LarKC), 2009.

[10] E. Gabrilovich and S. Markovitch. Wikipedia-based semantic interpretation for natural language processing. *Journal of Artificial Intelligence Research*, 34(1):443–498, 2009.

[11] Georgina Gallizo, Mick Kerrigan, Barry Bishop, Spyros Kotoulas, Luka Bradesko, Matthias Assel, Alexey Cheptsov, and Vassil Momtchev. D5.3.2 - Overall LarKC Architecture and Design v1, September 2009. Available at http://www.larkc.eu/deliverables/.

[12] Perry Groot, Heiner Stuckenschmidt, and Holger Wache. Approximating description logic classification for semantic web reasoning. In *ESWC*, pages 318–332, 2005.

[13] B.N. Grosof, I. Horrocks, and R. Volz. Description logic programs: combining logic programs with description logic. *Proceedings of the 12th international conference on World Wide Web*, pages 48–57, 2003.

[14] Z. Huang, F. van Harmelen, and A. ten Teije. Reasoning with inconsistent ontologies. In *Proceedings of the International Joint Conference on Artificial Intelligence - IJCAI'05*, 2005.

[15] Zhisheng Huang and Frank van Harmelen. Using semantic distances for reasoning with inconsistent ontolgies. In *Proceedings of the 7th International Semantic Web Conference (ISWC2008)*, 2008.

[16] Zhisheng Huang, Frank van Harmelen, Stefan Schlobach, Gaston Tagni, Annette ten Teije, Yi Zeng, Yan Wang, and Ning Zhong. D4.3.2 - Implemented Plug-ins for Interleaving Reasoning and Selection of Axioms, March 2010. Available at http://www.larkc.eu/deliverables/.

[17] U. Hustadt, B. Motik, and U. Sattler. Reducing SHIQ- Description Logic to Disjunctive Datalog Programs. *Proc. of the 9th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2004)*, pages 152–162, 2004.

[18] Integrated rule inference system. `http://www.iris-reasoner.org/`, 2010. STI Innsbruck.

[19] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM a pragmatic semantic repository for OWL. In *Web Information Systems Engineering (WISE) Workshops*, page 182192, 2005.

[20] M. Krtzsch, S. Rudolph, and P. Hitzler. Elp: Tractable rules for owl 2. In *Proceedings of the 7th International Semantic Web Conference*. Springer, 2008.

[21] O. Lassila, R.R. Swick, et al. Resource Description Framework (RDF) Model and Syntax Specification. 1999.

[22] Daniel D. Lee and H. Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 1999.

[23] Hansjorg Neth, Lael J. Schooler, Jorg Rieskamp, Jose Quesada, Jie Xiang, Rifeng Wang, Lijuan Wang, Haiyan Zhou, Yulin Qin, Ning Zhong, and Yi Zeng. D4.2.2 - analysis of human search strategies, September 2009. Available from: http://www.larkc.eu/deliverables/.

[24] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. Marvin: Distributed reasoning over large-scale semantic web data. *Web Semantics*, 7(4):305–316, 2009.

[25] Owl 2 web ontology language profiles. `http://www.w3.org/TR/2009/CR-owl2-profiles-20090611/`, June 2009. W3C Candidate Recommendation.

[26] Jose Quesada, Yi Zeng, Ralph Brandao, Lael Schooler, Stefan Otte, Yan Wang, Zhisheng Huang, Ning Zhong, and Danica Damljanovic. D2.3.2 - Cognitive Memories component v. 2. Subsetting by statistical semantics and user interests, March 2010. Available at http://www.larkc.eu/deliverables/.

[27] Jose Quesada, Yi Zeng, Lael J. Schooler, Haiyan Zhou, Ning Zhong, Yulin Qin, Shengfu Lu, Yiyu Yao, and Yang Gao. D2.3.1 - Cognitive Memories Components v1, March 2009. Available at http://www.larkc.eu/deliverables/.

[28] Rdf semantics. `http://www.w3.org/TR/rdf-mt/`, February 2004. W3C Recommendation.

[29] M. Sahlgren. An introduction to random indexing. In *Methods and Applications of Semantic Indexing Workshop at the 7th International Conference on Terminology and Knowledge Engineering, TKE 2005*. Citeseer, 2005.

[30] Marco Schaerf and Marco Cadoli. Tractable reasoning via approximation. *Artificial Intelligence*, 74(2):249–310, 1995.

[31] S. Schlobach, E. Blaauw, M. El Kebir, A. ten Teije, F. van Harmelen, S. Bortoli, M. Hobbelman, K. Millian, Y. Ren, S. Stam, P. Thomassen, R. van het Schip, and W. van Willigem. Anytime classification by ontology approximation. In Ruzica Piskac et al., editor, *Proceedings of the workshop on new forms of reasoning for the Semantic Web: scalable, tolerant and dynamic*, pages 60–74, 2007.

[32] Heiner Stuckenschmidt. Partial matchmaking using approximate subsumption. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, 2007.

[33] H. J. ter Horst. Combining rdf and part of owl with rules: Semantics, decidability, complexity. In *Proceedings of the 4th International Semantic Web Conference*. Springer, 2005.

[34] Volker Tresp, Yi Huang, Markus Bundschus, and Achim Rettinger. Materializing and querying learned knowledge. In *Proceedings of the First ESWC Workshop on Inductive Reasoning and Machine Learning on the Semantic Web*, 2009.

[35] Volker Tresp and Kai Yu. Learning with dependencies between several response variables. In *Tutorial at ICML 2009*, 2009.

[36] J.D. Ullman. *Principles of Database Systems*. WH Freeman & Co. New York, NY, USA, 1983.

[37] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.

[38] H Wache, P Groot, and H Stuckenschmidt. Scalable instance retrieval for the semantic web by approximation. *Lecture notes in computer science*, 3807:245, 2005.