

RDF Gears, a data integration framework for the Semantic Web

Master's Thesis



Eric Feliksik

RDF Gears, a data integration framework for the Semantic Web

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Eric Feliksik
born in 's-Gravenhage



Web Information Systems
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
<http://www.wis.ewi.tudelft.nl/>

RDF Gears, a data integration framework for the Semantic Web

Author: Eric Feliksik
Student id: 1200348
Email: feliksik@gmail.com

Abstract

This thesis describes the design and implementation of RDF Gears, a data integration framework for the Semantic Web. The RDF Gears Language combines the Semantic Web technologies with the Nested Relational Algebra. It provides an expressive Domain Specific Language for the development of workflows integrating RDF data with other sources. It allows Semantic Web developers and researchers to work on new, domain specific algorithms without wasting time on the implementation details of data transformation, storage and optimization.

A web-based user interface is presented to create RGL workflows that are visualized with a graphical syntax.

An execution engine is developed to function as a workflow interpreter. It implements aggressive pipelining, lazy evaluation and other optimizations. A comparison with the Silk Framework shows that this first implementation is already quite efficient.

Thesis Committee:

Chair:	prof. dr. ir. G.J.P.M. Houben, Faculty EEMCS, TUDelft
University supervisor:	dr. ir. A.J.H. Hidders, Faculty EEMCS, TUDelft
Committee Member:	dr. A.E. Zaidman, Faculty EEMCS, TUDelft

Preface

This work is the result of well over a year of research. Although I could not work on it full-time, I probably did anyway. During many train trips I have been pondering over the language design. Implementation issues fascinated me while falling asleep and often occupied my mind before I opened my eyes in the morning. Some implementation optimizations were contrived in the shower. It was a captivating process.

I am grateful to my supervisor Jan Hidders. He inspired me to embrace a rather technical subject, after a much softer Master's curriculum, and I thank him for the patience and passion with which he supervised me. It was very motivating. The many discussions we had were not only indispensable for the results of this thesis, but they were also great fun! Furthermore I would like to thank Geert-Jan Houben, the chair of the research group, who introduced me to the field of the Semantic Web. This opened a whole new world to me and I enjoyed studying the challenges it poses.

This thesis marks the end of my university studies, a period in which I have grown both as an engineer and as a person. Many people contributed to that process. Whether they supported me or challenged me, I am grateful to all of them. In particular I thank my parents for the love and support they have given me, allowing me to do things my way. Finally I am very thankful to my many friends for their invaluable support and warm friendship.

Eric Feliksik
Delft, the Netherlands
September 21, 2011

Contents

Contents	v
List of Figures	vii
1 Introduction	1
1.1 Motivation	1
1.2 Research questions	2
1.3 Contributions	4
1.4 Organization of this thesis	4
2 Background	7
2.1 Concepts	7
2.2 Related work	11
2.3 Conclusion	15
3 RDF Gears Language	17
3.1 Informal introduction	17
3.2 Notational conventions	19
3.3 RGL Data model	20
3.4 RGL Syntax	22
3.5 RGL Semantics	25
3.6 RGL core function definitions	27
3.7 Evaluation	34
3.8 Conclusion	42
4 Graphical User Interface	43
4.1 User interface overview	43
4.2 Produced XML format	43
4.3 Technical architecture	45
4.4 Evaluation	46
4.5 Conclusion	47
5 Implementation of the RDF Gears Engine	49
5.1 Command line interface	50

5.2	Basic architecture	50
5.3	Implementation optimizations	54
5.4	The implementation of RGL core functions	62
5.5	Extending RGL with custom functions	65
5.6	Evaluation of the implementation optimizations	66
5.7	Conclusion	71
6	Case study: Comparison with Silk	73
6.1	Silk introduction	73
6.2	A typical Silk transformation	74
6.3	Equivalent transformation in RDF Gears	74
6.4	Expressivity comparison	75
6.5	Performance comparison	77
6.6	Conclusion	80
7	Conclusion and Future Work	81
7.1	Summary	81
7.2	Future work	84
	Bibliography	87
A	Report of an earlier RGL design attempt	93
B	RGL XML syntax	95
C	Performance measurement data	97
C.1	Evaluation of the optimizer	97
C.2	Evaluation of Field Index Mapping	99
C.3	Performance comparison with Silk	101

List of Figures

2.1	The Linked Data cloud	8
3.1	An example workflow aggregating Linked Data about malaria	18
3.2	Some RGL query rewriting rules	41
4.1	A screenshot of the RDF Gears GUI	44
4.2	The workflow <code>silkGroup/minimumScore</code>	44
4.3	RDF Gears UI code and its relation to third party libraries	46
5.1	RDFGears engine options	49
5.2	A typechecking error is detected by the engine	50
5.3	Class diagram of the RGL Value class hierarchy	51
5.4	Class diagram of the RGL Value class hierarchy	52
5.5	Class diagram of the RGL Function	53
5.6	A simple workflow that greatly benefits from pipelining	55
5.7	A workflow that benefits from lazy evaluation	60
5.8	Example of using the bag categorize function, inspired by Silk.	63
5.9	Class diagram for the SPARQLFunction implementation	64
5.10	A SPARQL query that may yield more than 2500 results	64
5.11	The <code>testStream</code> workflow used for the pipelining experiment	67
5.12	The optimizer reduces both execution time and memory consumption. . .	68
5.13	Field Map Indexing reduces execution time and memory usage	70
6.1	Workflow equivalent to the Silk <code>linkedmdb_directors</code> example.	76
6.2	The trivial record projection workflow used by the top-score processor. . .	77
6.3	The workflow that categorizes a score as "accept", "verify" or "reject". . .	77
6.4	Execution time comparison between Silk and RDF Gears.	78
C.1	Memory usage during workflow execution	98
C.2	Silk LSL file	103

Chapter 1

Introduction

1.1 Motivation

In recent years the Semantic Web has been growing rapidly. The vast amounts of structured data available in the world are being converted to RDF data [23, 4] and published via standards such as Linked Data [1] or SPARQL endpoints [20]. By providing the data in a standardized format with globally unique identifiers for every entity, the Semantic Web technologies aim to facilitate data integration at web scale. The data is structured in a way that allows computer systems to reason over them, which can greatly benefit the user experience. In the traditional Web the user is largely responsible for finding, merging and summarizing relevant data on the Internet. The ultimate goal of the Semantic Web is to move these tasks to computer systems that provide the user with the information considered relevant, in the presentation form desired [5].

However, there are still many challenges to overcome. Making practical use of this data is far from simple as the data is often fragmented over different sources, accessible via different mechanisms (Linked Data, SPARQL, RDFa, JSON, XML, still in Relational Databases, on Twitter and other social media, published with custom API's, etc.) and data is incomplete and dirty. If data is in RDF format or can be converted to RDF, links between data sources are often missing and different ontologies are used. So although the availability of the data in RDF is a useful first step, real data integration requires more: to combine the data residing at different sources and provide the user with a unified view [34], in practice the different data sources must be transformed, manipulated and crafted to fit together. This often requires a lot of custom programming as the ways in which datasets must be transformed are domain specific and dataset specific. Furthermore, the actual goal is often to create a mashup or perform data mining on the dataset. Developing data mining algorithms is subject to trial and error. As it is often impossible to create an integrated dataset that answers to all needs in all scenarios, the data integration approach must be pragmatic and may be closely linked to the requirements of the mining algorithm. For this reason, a datamining process at web scale will often perform data integration steps itself.

Tools have emerged to facilitate execution of specific tasks, such as converting datasets from relational to RDF datasets [6, 3, 48], interlinking datasets [29, 24], federated querying [40, 44, 22] and domain-specific algorithms performing, for example, data mining [13, 42]. The specific tools perform valuable tasks for the direct con-

sumers of RDF data, often coming from a scientific or technical environment (e.g. from the field of bio-engineering). As one size does not fit all, practical data integration challenges constantly require new algorithms and techniques to make the data of practical use. Scientists working with large datasets on domain specific problems often have deep knowledge of the datasets they are working with and how these relate to each other. They often can come up with transformation steps to integrate or transform datasets and they may have some limited programming experience. However, if the functionality of a standard tool can no longer be stretched to fit a use case it was not designed for, custom programming work is again needed to modify or replace it. This is not only time-consuming, but the in-depth programming knowledge that is required is often *not* available.

This thesis attempts to provide a data integration framework that allows expressing and executing complex data integration algorithms without the need to be overly concerned with implementation details. Semantic Web data integrators and researchers can benefit from a tool that is more generic than the specific tools mentioned before. It provides more expressivity and flexibility, allowing the implementation of novel data integration mashups. At the same time it is more high-level and domain-specific than the already available high-level programming languages such as Java and Python. This makes it easier to understand and to get things done quickly.

1.2 Research questions

The main research question is as follows:

Is it possible to develop a framework for RDF data integration that is expressive, useful and can be implemented efficiently?

By framework we mean a domain specific language with a formal definition, a mechanism to extend this language, a concrete syntax to create expressions in this language, and a software system to evaluate these expressions.

The following sub-questions articulate the problem:

1. ***Is it possible to design a data transformation language and formally define its syntax and semantics?***

The syntax and semantics should provide a formal, unambiguous and complete language specification. Here we are concerned with an abstract syntax, by which we mean a specification of the syntax's key concepts. As opposed to a concrete syntax, it will not be concerned with the details of a particular serialization or graphical notation style. This is important to support the design of the workflow interpreters, compilers, optimizers and debuggers [50].

2. ***What is the expressive power of that language?***

The expressive power of a language is the extent to which ideas or operations are expressible. E.g., propositional logic has less expressive power than first-order logic, as the former does not allow the use of quantifiers (denoted with $\exists$ and $\forall$ in first-order logic). The expressive power can be assessed by showing that the language can simulate an existing formal language, or by showing how

an RGL expression can execute data transformations currently performed by existing products. We will do this for DERI Pipes and Silk.

3. ***What concrete syntax can be used for that language, and how can a user interface be implemented for it?***

With concrete syntax we mean a particular notation of the abstract syntax. This can be a serialization or a graphical notation style. These thus allow the user to create language expressions. The user interface should help the user to create and modify expressions in a convenient way.

4. ***Is the language useful for scientists and developers using Semantic Web data?***

Target users are Semantic Web developers and researchers with knowledge of the Semantic Web technologies SPARQL and RDF. We do not intend to create a system for the average computer user. The notion of usefulness is subjective and is eventually to be evaluated by or with the target users. Doing a full study on the usefulness is not within the scope of this thesis and therefore a complete answer can not be given. However, we must somehow make credible that the language is useful to the target group, otherwise there is no point in designing it. I will use the following three criteria to evaluate the usefulness:

- a) The language is expressive enough to express custom data integration algorithms. This may partially follow from the formal expressivity analysis, but is also answered with practical examples of data integration steps implemented in this language.
- b) The language provides a ‘right’ level of abstraction to make the expression of such algorithms convenient. Although the term ‘right’ is not sharply defined, it does point out that this is a balancing act. The language should not be as generic as a general purpose programming language, as this requires the user to deal with implementation details. On the other hand, if the language is too specific, this will limit the flexibility and expressivity.
- c) The functionality is clearly defined and understandable. If it is not, the language will be hard to learn, hard to use and unsuitable to exchange or communicate algorithms between researchers.

5. ***How can an efficient execution engine for that language be implemented?***

An execution engine is a program that takes an expression as input and evaluates it according to the language semantics specification, to finally output the result. The efficiency of the implementation is determined by the amount of processing time and memory space required to evaluate a given language expression. This can be evaluated with measurements on the memory and run time requirements of an actual implementation, but also by showing that the language is suitable for the application of well-known optimization techniques from theory. Examples are database query optimization techniques that reduce evaluation complexity and techniques like parallelism and pipelining. As such techniques are not applicable to all languages, the applicability indicates a language quality.

1.3 Contributions

The main contribution of this research is the RDF Gears framework. It can be subdivided in three separate contributions:

- The design of the RDF Gears Language (RGL). It is a data integration language for which the syntax and semantics are formally defined.
- The design and implementation of the RDF Gears graphical user interface (GUI). It is a prototype based on the code of DERI Pipes [36] and allows convenient construction of complex expressions in RGL.
- The RDF Gears Engine. The engine allows the evaluation of RGL expressions and thus performs the data integration. It shows that an efficient implementation of RGL is possible.

RDF Gears can serve as a Swiss Army knife for developers and researchers working in the field of applied RDF data integration. It allows them to express and execute complex data transformation algorithms without being concerned with the technical details of implementation. This way they can focus on the domain specific algorithms and applications they care about, which in turn can accelerate the research and development that solves real-world problems.

As RGL is based on well-known formalisms like Set Theory and the Nested Relational Algebra and applies these to practical technologies like RDF and SPARQL, it provides an interesting platform for research on the relations between these technologies, e.g. for the development of query optimization techniques.

In the process of designing and implementing RDF Gears I have been active on some mailing lists of Open Source software projects like Jena/ARQ and Silk. Some of the suggestions and bug fixes I contributed there have been picked up upstream (e.g. [46, 47]).

1.4 Organization of this thesis

The structure of this document is as follows. First, chapter 2 introduces the Semantic Web, the concept of data integration and of workflow systems. It also discusses related work that is implemented in software.

Chapter 3 presents the design of the RDF Gears Language. It first presents the graphical syntax and an informal introduction of the semantics by means of an example that aggregates Linked Data. This is supposed to make the rest of the chapter more comprehensible. Then some notational conventions are defined, after which the formal language specification is given: it defines the data model, the RGL abstract syntax, as well as the semantics of the workflows and of the core functions. Then the design rationale and an evaluation of the language expressivity, usefulness and optimizability is presented.

Chapter 4 presents the user interface. It presents a brief overview of the functionality. It discusses the XML format in which the workflows are serialized, and how these documents are stored. The technical architecture is described, explaining the compo-

nents RDF Gears is built upon. An evaluation assesses the strengths and weaknesses of the user interface.

Chapter 5 describes the engine implementation in detail. First the engine executable and its command line options are presented. Then the engine architecture is presented by explaining the basic structure of values, functions and workflows and the typechecking and execution mechanism. Different implementation optimizations are discussed, among which are the pipelining architecture, the optimizer that determines where intermediate values must be materialized in memory, the lazy evaluation mechanism and the Field Map Indexing technique. We discuss some function implementations to show how these fit in this architecture, and finally evaluate the optimizations to show that these significantly reduce the memory usage and execution time of workflows.

Chapter 6 is a case study that implements a typical data integration algorithm of the Silk framework. It shows how an example data integration algorithm from the Silk distribution can be implemented in RDF Gears, and where RGL provides expressivity that Silk lacks. A performance evaluation compares RDF Gears with Silk.

Chapter 7 concludes this thesis. It provides a summary with conclusions and discusses future work.

Chapter 2

Background

This chapter discusses background concepts as well as concrete tools that are relevant existing work. Section 2.1 first introduces the Semantic Web technologies such as RDF, Linked Data and SPARQL. Then the concepts of data integration and workflow systems are explained. Section 2.2 provides an overview of the most relevant existing work, and section 2.3 concludes the chapter.

2.1 Concepts

2.1.1 Semantic Web

The traditional Web, invented by Tim Berners-Lee in 1989, is a web of documents. The HTTP protocol defines a way to retrieve a document, if the identifying URI is known. The document can refer (or *link*) to other documents by means of a URI. The HTML documents are machine-readable structures that both describe the data that should be shown to the user (content) as well as how the data should be displayed (lay-out). A web page can for example contain the text "The capital of The Netherlands is the city Amsterdam". For a human user this is useful text. To a computer, however, data on the Web is unintelligible: the data is often unstructured, natural language. Even the content of tables contained in the document is merely a set of characters to the machine displaying it, as the computer system does not know what is described.

The Semantic Web, on the contrary, is a web of data. It is not supposed to replace the current Web, but rather add a layer of functionality. The idea is to publish data in such a format that computer systems are able to process the described data and reason over the relations between different data elements. For this, a number of technologies are useful. RDF is used to represent data, and SPARQL is used to query it. This thesis focuses on using these two technologies to create an even more powerful data transformation language. Other technologies used in the Semantic Web, like the OWL language that is used to model ontologies, are beyond our scope.

RDF¹, short for Resource Description Framework, is the data format used on the Semantic Web. It consists of triples like²:

`<http://dbpedia.org/resource/Netherlands> <http://dbpedia.org/ontology/capital>`

¹<http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>

²The N-Triples format is used to denote the RDF triple here; see <http://www.w3.org/TR/rdf-testcases/#ntriples>



Assuming that the URI's are understood to refer to *The Netherlands*, the concept of *having a capital city*, and the *city of Amsterdam*, respectively, this triple states that Netherlands has the capital Amsterdam. Many triples together form a graph of data. Different collections of triples (different graphs) can use the same URI's to refer to concepts. Thus the graphs are interconnected and a Web comes about.

Linked Data³ is a mechanism to publish RDF data via HTTP. The idea is to make the URI's that identify entities dereferencable. HTTP requests are forwarded to a location where an RDF document describing the topic can be download. It is a simple way of finding information about a particular entity and to find links to other datasets. As many datasets have been created in recent years and these datasets contain RDF links to one another's entities, a web of these datasets emerged. This web is called the Linked Data cloud, depicted in figure 2.1. The small circles are datasets, and the lines between them are links. As can be seen, DBpedia is often linked to other datasets (due to its very broad scope) and is thus the de-facto center of the Linked Data cloud.

⁴<http://www.w3.org/TR/rdf-sparql-query/>

An example of a SPARQL 1.1 SELECT query that can be executed on an endpoint is:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpedia: <http://dbpedia.org/ontology/>
SELECT ?label ?mov
WHERE {
  GRAPH <http://dbpedia.org/> {
    ?mov rdf:type dbpedia:Film.
    OPTIONAL {
      ?mov rdfs:label ?label.
    }
  }
}
```

This query would select all Film entities of the graph `http://dbpedia.org/` and include, if available, the label for every movie. It would thus return a list of records that contain the *mov* and *label* keys, which are mapped to the URI identifying the movie and the label of that movie.

2.1.2 Data integration

Although the availability of RDF data is very useful to combine it with other datasets, there is still a lot of work to do. Data integration is the problem of combining data residing at different sources, and providing the user with a unified view of these data [34, 18]. Imagine datasets *A* and *B* are published in RDF. However, the data still resides at different locations. If the datasets are merged they may both contain data about the same entities, but this may not be clear when different URI identifiers are used. If these entity identifiers are made uniform or identifiers are explicitly identified as so-called co-references, the datasets may still describe their miniworlds using different ontologies.

Simply merging the datasets *A* and *B* is possible for small datasets. For large datasets, it is often undesirable: it not only increases the total data size, but also leads to consistency issues if one of the original datasets is updated. If both datasets are publicly accessible, an approach can be to create links between the dataset and create a dynamic wrapper mechanism that provides a unified view, e.g. via a SPARQL endpoint. This is what federated query mechanisms [40, 44, 22] intend to achieve. Dynamic wrapper approaches deal with performance challenges and often use very specific algorithms to index and join the data. This may mean that this task is hard to generalize in a workflow system, and is instead better executed by highly specialized tools. Simply merging the datasets, as well as finding the correspondences between datasets, is done in a batch way and therefore less time critical. High level languages that facilitate this can be helpful to fulfill this task.

The challenge of identifying the semantically identical entities in dataset *A* and *B* is often called instance matching, interlinking, or the finding of co-references. Although ideally different datasets would all use the same URI for the same entities, in practice this is not the case: there is no central authority, and thus anybody can make up a URI to identify his subject. Also, as RDF data is often generated from other sources, the URI's are not consciously designed for every entity, but instead generated from the dataset that is being published. So the URI's in different datasets that identify the

same real world entity must be found. There are different ways to do this, but most approaches boil down to a fuzzy matching algorithm that tests all combinations of datasets A and B . Formally, the set of links is then defined [19] as

$$M = \{(a, b) | \text{sim}(a, b) \geq \theta, a \in A, b \in B\} \quad (2.1)$$

where *sim* is some kind of similarity function determining the probability that a and b are the same entities and θ is a threshold. The recall and precision will vary for different datasets, similarity functions and thresholds. We will briefly discuss the tools Silk [29] and LinQuer [24] in the next section, and more tools exist. Sometimes the algorithm these tools implement is too rigid, being very similar to equation 2.1. It is hard to modify that algorithm at a conceptual level without touching the programming code. Also, the tools may not assist in accessing the different data sources, as described above. For example, if dataset C is already interlinked with B , the process of interlinking A and B may be helped by involving data from C in the *sim* function. This is something not supported in the aforementioned tools and they can not be easily extended to access a third dataset. A more flexible, generalized language may help here.

If the interlinking process is fulfilled, there is still the problem of *ontology alignment* or *ontology mapping* [30, 18, 54]. This is the task of finding what properties and instance types in dataset A correspond to what properties in dataset B . It is often considered as a research subject separate from the problem of instance matching, but some approaches (e.g. [43, 17]) combine the problems in an attempt to achieve better results.

In practice it often happens that the mappings can not simply identify one-to-one correspondences between ontologies, but that they are more complex [21]. For example, dataset A models an address as a single string, whereas ontology B provides a more granular approach. In dataset B the street name may be provided as a string, the address number and zip code are separately given, and the city name is identified by means of a URI. In other cases, conditional mappings are required. In such cases it is practically impossible to perform mappings and instance matching automatically, and a lot of manual tweaking is required. Although some mapping languages provide the expressivity for complex mappings (e.g. the languages mapping relational databases to ontologies [33, 7]), these tools often focus on dynamic wrapping. As a result, the types of datasources that can be accessed as well as the types of mappings that can be expressed (the mapping language expressivity) are limited, to make query translation feasible and to allow for good dynamic performance. This is very useful, but in practice the properties of Semantic Web datasets (different access methods, dirtiness, incompleteness, widely different ontologies requiring complex mappings) make that the specialized tools alone are insufficient to achieve the data integration task. A lot of manual crafting, preprocessing of datasets and manual mapping construction is required. Even though a high-level language to operate on RDF data will not be able to replace the specialized tools, it can be used as glue code to combine them. Also, they can be used to implement highly customized algorithms that can not be implemented with existing, more specific tools.

2.1.3 Workflows

Scientific workflow systems are systems that offer a set of high level operators that all do a single well-defined specific task, which can be composed in a larger whole – the workflow. Like in Unix, piping can be used to flexibly combine simple operations that process each others output. For example, the command

```
$ cat foo.txt | grep "bar" | sed "s/fantastic/lovely/g"
```

filters those lines in a file that contain the string "bar" and then, in those lines, replaces all occurrences of "fantastic" by "lovely". A scientific workflow system also applies this piping paradigm. However, a scientific workflow often supports loops and uses an acyclic directed graph structure for the forking and merging of data streams. The acyclic directed graph nature of the workflows makes them suitable for a graphical representation, and many workflow systems feature such a syntax and interface.

Whereas many workflows are concerned with the timing, concurrency, and synchronization aspects of a complex process, these are not important in the goals of this thesis. It cares more about the data that is transformed. Some authors prefer to call such systems dataflows [26], and this thesis is by that definition actually presenting a dataflow framework. However, the term workflow system will be used loosely from now on and is thus also used to describe the result of this research.

Existing workflow systems have shown that high-level, domain specific languages based on a functional paradigm can provide constructs that closely match the way users think about data transformation [37]. But more than that, it allows automatic optimizations such as parallelized distribution of the data integration process. Data integration algorithms are often of quadratic complexity and the datasets keep growing, requiring more and more computing power. At the same time, the trend of Moore's law has come to an end. This should make clear that optimization is important – but as it is a means to an end, it is good to not bother the user with this.

This optimization requires a lot of attention in the procedural implementations, which must for example be manually fit into a parallelization framework such as Map Reduce [16]. On the contrary, scientific workflows often have a functional nature. This makes it possible to map the algorithms to a Map Reduce framework automatically. This allows the user to forget about such implementation details and to focus on the algorithm itself, while the performance optimizations that are necessary to make the computation feasible are still achieved.

2.2 Related work

It is impossible to discuss all existing work in the field of data integration in the Semantic Web. Also, there are many previous approaches that aim to create a high-level domain-specific programming language – some for RDF data and some more generic. The known work that is considered the most relevant is discussed below. Some approaches are selected because they provide a high-level, workflow-like, domain-specific programming language. They may or may not operate on RDF data. Other approaches may not be generic solutions, but are selected because they perform some specific transformation or integration step on RDF data.

Taverna

Taverna [27] is a scientific Workflow Management System that is widely used in the field of bio-informatics research. It is used to deal with large amounts of data it can fetch from bio-informatics Web services. As Taverna focuses on the orchestration of such Web services, the dataflow syntax offers constructs to deal with the stateful nature and possible side-effects of these services. It is an Open Source system with a large community of researchers that share workflows.

Taverna 1 was originally developed to support the practical need of a wide variety of users and to automate their integration tasks. It evolved to a serious workflow management system with a proven record of usefulness in the field of bio-informatics. Also, it does support the wider context of facilitating the discovery and reuse of workflows well. The Taverna Workbench provides a desktop program to construct workflows and integrates this with an execution engine. The engine is also available as a separate program for the command line, or as a server application. The Taverna 1 language lacked a formal definition that made optimizations difficult. Sroka et al. [50] developed a scientific underpinning of the Taverna 2 language syntax and semantics and gave an unambiguous and complete specification thereof. This newer version now implements optimization techniques like streaming and parallel execution, making it possible to process large datasets on computational grids and clouds with the Taverna server engine.

Taverna does not natively support Semantic Web technologies like SPARQL. Its string-based data model is not the most suitable for RDF data. Also, it does not focus on data integration, but rather on the orchestration of Web services. The requirements for these use cases do have some similarities, but our data integration goals benefit from tight integration with Semantic Web technologies. The data integration requires less emphasis on the stateful nature and possible side-effects of Web services.

Oinn et al. [37] discusses the lessons Taverna 1 taught on how to create a workflow environment for the life sciences. The first one is *abstraction is key*; the Taverna pipeline syntax corresponds to the conceptual way people think of the dataflow, and it does not deal with implementation details. The second is *Finding the limits of abstraction*: too much abstraction takes away the power from the user. When people become more skilled in the use of the workflow system, they push its representation as they want to perform more complex tasks (e.g. special forms of iteration, conditional branching). The authors argue that there is a tension between keeping the language simple for the majority of users, and supporting more advanced functionality. Other lessons focus on the heterogeneous nature of Web services in the field of bio-informatics, and that it is important that a workflow system can deal with the different protocols and data models. In our context this is just as relevant: RDF data may be published via Linked Data, SPARQL endpoints, RDF dumps, and my need to be combined with data from relational databases, Twitter posts, Facebook profiles, etcetera. Another lesson is the importance of the 'wider context', the support of the user community by making it easy to find, reuse, extend and share workflows. The authors argue that these lessons are also relevant to other areas of data intensive and exploratory science. This seems very reasonable and suggests they also apply to our data integration framework for the Semantic Web.

Kepler

Kepler [35] is a scientific workflow management system that aims at orchestration of bio-informatics Web services. It incorporates the required control flow mechanisms for such services and offers the ability to execute workflows on the grid. During the evolution of the system it became apparent to the developers that concepts of the world of functional programming, such as the *map* operation, are very useful in a workflow system. The functional programming constructs not only offer good high-order constructs that conceptually fit the workflow paradigm well, but also are also well-known and easy to understand, making optimization more feasible. It is a robust system with a large user base, but its focus on data model Web service orchestration and its data model make it not particularly suitable for RDF data integration.

Pig Latin

Pig Latin [38] is a dataflow language that uses a nested data model similar to the Nested Relational Calculus and uses the primitive types `int`, `long`, `double`, `chararray` and `bytearray`. The Pig implementation features Map Reduce based parallel processing. Pig Latin describes a dataflow similar to a workflow system, but does not offer a graphical syntax or user interface. Instead, the syntax is based on a series of statements that bind local variables, which can to a certain extent be reordered by a query optimizer.

RDF data structures are not natively supported, and Pig does not offer any facilities to operate on such data. The authors of RAPID+ [41, 49] note that RDF graphs can be simulated in bags of 3-tuples, but that the Pig Latin operators are not as effective for processing complex analytical queries on RDF. RAPID+ therefore implements extensions to Pig Latin, partially overcoming this limitation and using Pig to process RDF data. Although this is promising, it is not clear whether a graphical representation of the syntax would be as intuitive as that of a system that is designed as workflow system from scratch.

Silk

Silk [29] is a framework providing tools to generate links between data items based on user-provided link specifications. Unlike the bio-informatics workflow systems it specifically aims on Semantic Web data and the integration thereof. In the past years large amounts of bio-informatics data came available in the Linked Open Data cloud, which has inspired the Silk authors to give examples of how these datasets can be integrated with Silk.

The user specifies fuzzy matching rules that are evaluated for pairs of elements of two different datasets. It can be used by data publishers to generate links between datasets as well as by Linked Data consumers to extend Web data with additional RDF links. Silk thus interlinks these datasets and focusses on parallelism and optimization. It does take the *abstraction is key* lesson of Taverna to heart by providing a concise XML-based language for interlinking tasks. However, this language aims to express one type of data integration task and it is thus not a generic workflow system. This way it loses expressivity. Although it is easy to use for beginners, this makes it unsuitable for a wide variety of data integration scenarios like we aim to support. So for some

purposes, Silk overstretches the *limits of abstraction*. Silk will be further discussed in our evaluation in section 6.

CloudFuice

CloudFuice [51] aims to use cloud computing to do mashup-based data integration. It introduces a scripting language that actually represents a workflow syntax. It uses a datamodel that features records and bags (although it is not clear whether these can be nested). The CloudFuice operators show the influences of the functional programming world: CloudFuice offers set operations such as union, intersection and difference, and the map and filter operations. Descriptions of these functions are given in [52]. Furthermore, operators to fetch remote data are provided. Although RDF data can be fetched, it is not clear how this data is further dealt with. The authors do not emphasize the language design, expressivity or usability. Instead they show that the data integration process can be executed in the cloud. Details are provided on how CloudFuice determines whether datasets are partitionable and how partitioning takes place.

A mechanism is provided to generate database queries based on data results earlier in the workflow, useful to query entity search engines. The prototype implementation uses a combination of Bigtable for data storage and Google Spreadsheet for the specification of the CloudFuice scripts and input data. Operators and tasks are implemented as REST-based Web services and use JSON as data exchange format. This way, custom operator can be implemented in any language. It is not clear whether this seemingly costly operator communication mechanism is the only one, and how this influences the performance when custom functions are executed repetitively.

DERI Pipes

DERI Pipes [36] is a visual tool to create data integration mashups. Like some other mashup tools it is inspired by Yahoo Pipes. It provides operators to work on formats like XML, RSS, XHTML, RDF and JSON. These operators can be combined in a workflow to describe a larger data transformation, thus integrating data on the Web.

DERI Pipes does not aim at integration of large datasets. The implementation is not designed to prevent repetitive work and the language is not formally defined. The lack of a clear data model and fundamental underpinning seems to lead to the ad-hoc creation of operators for every piece of functionality. This makes it harder to understand, as well as to optimize and evaluate the language. Even though SPARQL construct queries may be used to simulate expressive languages like NRC, it makes the system practically unsuitable for very complex transformations. However, the workflow-like structure with emphasis on Web technologies is inspiring for Semantic Web data integration.

LinQuer

The LinQuer framework LinQuer [24] is a system for generating SQL queries for semantic link discovery over relational data. LinQuer uses an SQL-like language called LinQL to specify linkage requirements. It offers a web interface and an API that translates this language into actual standard SQL queries, and also features an interface that

assists users in writing LinQL queries. It is very similar to Silk, offering the user a way to match different properties of entities in a source and a target dataset, to determine the similarity. It is thus not a generic workflow system, but instead has a narrower focus and a more limited expressivity. LinQuer was used to link LMDb to DBpedia [23].

SPARQLMotion

SPARQLMotion [53] is a commercial package for the processing of Semantic Web data. It features a graphical scripting language similar to that of scientific workflow systems that allows the user to describe queries, data transformations and mashups. It also provides a graphical user interface to create and execute such data transformations, but the proprietary nature of the product makes it hard to assess its strengths and weaknesses.

2.3 Conclusion

The RDF data format is designed to provide a flexible data modeling language and facilitate web-scale data integration. The flexible triple-format allows modeling a domain of data structured with some ontology. The URI's used to identify entities provide globally unique identifiers, providing a means to reuse identifiers in different datasets. The RDF data is published as Linked Data or via a SPARQL endpoint, making it accessible for use in mashups or integration in other datasets.

However, really achieving data integration requires more than just publishing RDF data. Data integration is the problem of combining data residing at different sources, and providing the user with a unified view of these data. This requires combining RDF data residing at different sources published via different mechanisms. Also, duplicate entities that still have different URI's must be identified. This is called *instance matching* or *interlinking*. Furthermore, the datasets are likely to be structured according to different ontologies. The problems of *ontology alignment* (also called *ontology mapping*) deal with unifying the datasets on this level of structure, thus providing the last step in this unified view. All this is necessary before any actual analysis on the unified dataset can be performed.

Some existing work aims at performing a specific, well-defined task in the process of RDF data transformation and integration. Such tools are often well optimized for that specific step, but still require preprocessing of the data to satisfy the tools preconditions. This preprocessing requires other tools, while the specialized nature of the approach may still imply that its functionality is too limited for the task at hand.

It can be concluded that in practice, every RDF data integration process is different. There is no generic divide-and-conquer approach to tackle the problem. Even though very specific tools are very useful, the process can seldomly be completed with a single tool. A high-level workflow language to operate on RDF data will not be able to replace the specialized tools, but can be used as a glue code to combine them. Also, it can be used to implement highly customized algorithms that can not be implemented with existing, more specific tools.

There are approaches that provide a high-level, domain-specific programming language. Workflow systems provide such a language that can be visualized in an acyclic

directed graph structure. The approach not only offers a lot of flexibility in the algorithms that can be expressed, but the paradigm can also be understood by non-programmers dealing with a data integration process. Finally, the functional workflow languages can allow optimizations without requiring the user to consider these in the algorithm specification. Examples are database-like query optimizations, as well as mapping the data transformation to a parallelization framework such as Map Reduce.

However, there is no single workflow system that is generic enough to express a wide variety of algorithms, allows specifying this on a high level of abstraction, fits the RDF paradigm well, has a fundamental basis required to implement optimization techniques, and that provides an intuitive graphical syntax.

Chapter 3

RDF Gears Language

This chapter presents the RDF Gears Language (RGL). First, section 3.1 presents the graphical syntax and an informal introduction of the semantics by means of an example. This is supposed to make the rest of the chapter more comprehensible. Section 3.2 introduces some notational conventions used in the mathematics in the rest of this document. Section 3.3 introduces the data model and defines the RDF objects and their types. Section 3.4 defines the syntax with which workflows can be described. It also defines a type inference system to verify whether a workflow is well-typed. The semantics of the RGL workflows is defined in section 3.5, and the semantics of core functions is given in 3.6. Section 3.7 presents an evaluation of the design rationale, the expressivity and the optimizability of the language, after which the chapter is concluded.

3.1 Informal introduction

RGL is a workflow language that combines *processors* in a *workflow*. Processors are blocks that represent a certain *function*. The functions have named arguments. For every argument of a function, the associated processor has an *input port* named with the argument name. A processor has one output. By connecting the output of one processor to an input of another processor, it is possible to create an acyclic directed graph: the workflow. The edges model the dataflow, with values being passed from output ports over the edge to the input port of another processor. A workflow itself also represents a function. It can therefore be used as function for another processor, in a larger workflow. This way, workflows can be *nested*.

The basic data elements are *RDF Literals* and *URIs*, and *RDF Graphs* that contain triples of these elements. Values can be aggregated in complex types: a *Record* is a mapping of field names to values. A *Bag* is an unordered collection of values, possibly with duplicates. As Records and a Bags are values themselves, these can be nested in arbitrary ways: for example, a SPARQL SELECT query returns a bag of records.

Take for example the workflow of figure 3.1. Processors *proc_1* and *proc_4* nest workflows. As the data flows to the right side of the processors, the URI of processor *proc_0* is input into the port *obj* of processor *proc_1*. The processor *proc_1* receives this URI, and the custom FetchRDF function resolves the URI and downloads Linked Data from the Web. The resulting RDF graph is output to *proc_3*. The SPARQL

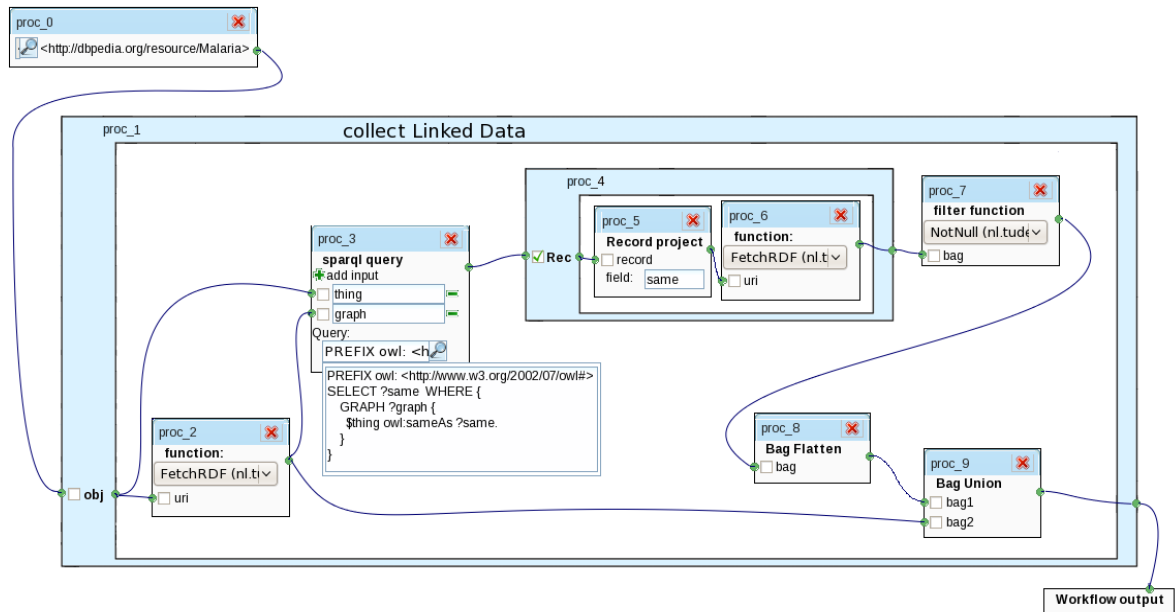


Figure 3.1: An example workflow aggregating Linked Data about malaria

function of the processor contains a query with the free variables *\$thing* and *\$graph*. As the input of port *thing* is a URI, the *\$thing* variable in the query is replaced with the URI value. The query is executed on the graph input in the *graph* port, producing a bag of records. Each record contains a field with the name *same*. This bag is input into *proc_4*. This nested workflow iterates over the input *Rec*, denoted by the marked box next to the input port. This means that the function of *proc_4* is executed on every element of the input bag, and that the results are collected in a bag which is output to *proc_7*. So *proc_5* is executed repetitively for every result of the SPARQL query, reading each record and outputting the value under the field with the name *same*. This value is assumed to be a URI, which is resolved to download Linked Data. The resulting RDF graph is output by *proc_6*, unless downloading fails – in that case, the *null* value is output.

As *proc_4* iterates, it will output a bag of graphs: one graph for every result of *proc_6*. The Filter function of *proc_7* uses the NotNull function. This function receives one argument and returns false if it is a *null* value. The NotNull function is used to test every element of the input bag of *proc_7*, to determine whether that value should be in the output bag. The BagFlatten function of *proc_8* receives a bag and merges all its elements. So a bag of bags would be merged into one big bag; in this case, the bag of graphs is merged into one big RDF graph. This big graph is then merged with the original RDF graph that was available at <http://dbpedia.org/resource/Malaria>. The result is a single RDF Graph which is output by the workflow. The workflow can be extended – for example, by attaching a SPARQL CONSTRUCT query at the end, which could unify the identifier used for malaria, using the *owl:sameAs* links.

Summarizing, the workflow collects Linked Data from the web by following all *owl:sameAs* links. It is an example of how the RDF Gears Language can be used to

perform a data integration step.

3.2 Notational conventions

Before proceeding with the formal definitions of the RGL data model, the syntax and the semantics, some notational conventions are introduced here.

$A \subseteq B$ denotes that A is a subset of B . A proper subset of B is written $A \subset B$. The set of all functions $f : A \rightarrow B$ is written B^A . For a function $f : A \rightarrow B$, we write $\text{domain}(f)$ to denote A , the domain of f .

If S is a set of elements, then a *bag* (or *multiset*) over a set S (or a bag of elements from S) is a function $B : S \rightarrow \mathbb{N}$ with $\mathbb{N}$ the set of natural numbers. For two bags B_1 and B_2 , both over the same set S , the union $B_1 \cup B_2$ is a bag $B_{1,2} : S \rightarrow \mathbb{N}$ such that $B_{1,2}(x) = B_1(x) + B_2(x)$ for all $x \in S$. We use a bag notation that we illustrate with an example: the notation $B = \{\{a, a, b, c\}\}$ is a bag with $B(a) = 2$, $B(b) = 1$, $B(c) = 1$ and for all $x \in S$ with $x \notin \{a, b, c\}$, $B(x) = 0$. We write $|B|$ to denote the number of elements in the bag, so in this example $|B| = 4$.

We can treat a function $f : X \rightarrow Y$ as a set of tuples $\{(x, f(x)) | x \in X\}$, and thus a bag B over S can be treated as a set $\{(x, B(x)) | x \in S\}$.

We postulate $\mathbb{K}$ as an infinite set of *field names*, and define the concept of a row as it is done in [10]:

Definition 3.2.1 (Row). A row over a set S is a mapping from a finite set of field names to S . It is a finite set of pairs.

If r is a row over S with the domain $K = \{A, B, C\}$, that is $r : K \rightarrow S$, and r is defined as $r(A) = a$, $r(B) = b$, $r(C) = c$, then we write r as $\langle A : a, B : b, C : c \rangle$.

If r_1 and r_2 are rows over S , and $\text{domain}(r_1) \cap \text{domain}(r_2) = \emptyset$, then $r_1 \cup r_2$ is a row over S with domain $\text{domain}(r_1) \cup \text{domain}(r_2)$, defined as follows:

$$r_{1,2}(k) = \begin{cases} r_1(k) & \text{if } k \in \text{domain}(r_1) \\ r_2(k) & \text{if } k \in \text{domain}(r_2) \end{cases} \quad (3.1)$$

Note that this definition of $r_1 \cup r_2$ is consistent with the set-union of the set-interpretations of r_1 and r_2 .

If B_1 and B_2 are bags over rows with disjoint domains K_1 and K_2 (that is, there are sets $K_1 \subset \mathbb{K}$ and $K_2 \subset \mathbb{K}$ such that $K_1 \cap K_2 = \emptyset$ and $\forall (r, c) \in B_1 : \text{domain}(r) = K_1$ and $\forall (r, c) \in B_2 : \text{domain}(r) = K_2$), then we define the Cartesian bag-product of B_1 and B_2 as follows (it is like the cross-join operation of SQL):¹

$$B_1 \overline{\times} B_2 = \{(r_1 \cup r_2, c_1 \cdot c_2) | (r_1, c_1) \in B_1 \wedge (r_2, c_2) \in B_2\} \quad (3.2)$$

The notation $t = (\cdot, \cdot, e, \cdot)$ means that t is a 4-tuple and that e is the third element of t .

¹Note that we used the symbol $\overline{\times}$ instead of $\times$, because the latter is the Cartesian product in set theory. The operator $\times$ can also be applied on the set interpretation of two bags, but would yield a set of tuples that could no longer be interpreted as bags. Furthermore note that the $\overline{\times}$ operator is commutative, unlike $\times$.

3.3 RGL Data model

The values of RGL combine the value system of the Named Nested Relational Calculus (NNRC) [10] with the RDF data model. A static type system is used, except for the distinction of RDF literals and URI's that cannot be distinguished statically as SPARQL queries may bind any of these types to a variable.

3.3.1 Values

We postulate the countably infinite sets $\mathbb{U}$ of *URIs* and $\mathbb{L}$ of *literals*. The sets $\mathbb{U}$ and $\mathbb{L}$ are pairwise disjoint and are further specified by the definition of RDF [31]. We say that $\mathbb{V} = \mathbb{U} \cup \mathbb{L}$ is the set of *RDF values*.

Definition 3.3.1 (RDF Graph). *An RGL Graph (or just graph) is a finite subset of $(\mathbb{U}) \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{L})$. We say that a graph $G \in \mathbb{G}$ contains an RDF value $v \in \mathbb{V}$ iff v occurs in any triple in G . That is $\exists(s, p, o) \in G : v=s \vee v=p \vee v=o$.*

Remark 3.3.2. *Note that blank nodes are currently not officially supported in RGL Graphs. The definition of Graph may be extended later, but this requires both careful analysis and implementation. The semantics of blank nodes in RDF is only defined in the context of a certain graph or SPARQL result. It is likely that a sensible semantics can be formulated, e.g. one that considers blank nodes as a tuple (i, r) with i the node identifier and r the SPARQL result set for which the identifier holds. The usefulness and limitation of such semantics in the context of RGL would then have to be considered. Such an extension of formal RDF semantics may conflict with existing RDF and SPARQL implementations.*

We define $\mathbb{G}$, the infinite set of graphs. We will now tailor the data model from the NNRC to fit the RDF paradigm.

Definition 3.3.3 (RGL Values). *The set $\mathbb{C}$ of RGL values is inductively defined as follows:*

- *every RDF value is a value*
- *a graph is a value*
- *if r is a row over values, then $[r]$ is a value that we call a record*
- *a finite multiset of values is a value that we call a bag*
- *the special element ω is a value that we call the null value.*

Also note that further on, Remark 3.3.7 will restrict bag values to contain only values of the same type.

Remark 3.3.4. *When the context makes clear that we are describing a value of the **Record** type, we can omit the row symbols $\langle \rangle$ in the notation of the record value. For example, the record $[\langle D : v_1, E : v_2 \rangle]$ with v_1 and v_2 arbitrary values can be simply written as $[D : v_1, E : v_2]$.*

Assume that $u_n \in \mathbb{U}$, $l_n \in \mathbb{L}$, $G_n \in \mathbb{G}$ for all n , and $A, B, C, D, E \in \mathbb{K}$. An example of a complex value is

$$[A : l_1, B : \{\{u_2, u_3, u_4\}\}, C : \{\{[D : G_5, E : l_6], [D : G_7, E : \omega]\}\}]$$

It should be clear by now that this is a record with fields A, B, C ; that if the record is called v , then $v.A$ is the Literal l_1 ; that $v.B$ is a bag with the URI's u_2, u_3, u_4 ; and that $v.C$ is a bag with the records $[D : G_5, E : l_6]$ and $[D : G_7, E : \omega]$.

3.3.2 Type system

RGL features a static type system that allows to evaluate correctness of expressions before execution. We introduce the types $\mathcal{U}$ for URI's, $\mathcal{B}$ for blank nodes, $\mathcal{L}$ for literals, $\mathcal{V}$ as an abstract supertype for RDF values of types $\mathcal{U}$, $\mathcal{B}$ and $\mathcal{L}$. We introduce the type $\mathcal{G}$ for RDF graphs. For the remaining types we use the definition of [10] modified to have bag-semantics instead of set-semantics:

Definition 3.3.5 (RGL Types). *The set of types in RGL is inductively defined as follows:*

- $\mathcal{V}$ and $\mathcal{G}$ are types
- if ρ is a row over types, then **Record**(ρ) is a type.
- if T is a type, then **Bag**(T) is a type.

Definition 3.3.6 (RGL Type Semantics). *The semantics $\llbracket T \rrbracket$ of a type T is the set of all values of that type. It is inductively defined as follows:*

- the semantics of the type $\mathcal{V}$ is $\llbracket \mathcal{V} \rrbracket = \mathbb{V} = \mathbb{U} \cup \mathbb{L} \cup \{\omega\}$, i.e. the set of all RDF values (but not blank nodes) and the null value.
- the semantics of the type $\mathcal{B}$ is $\llbracket \mathcal{B} \rrbracket = \mathbb{B} = \{\text{True}, \text{False}, \omega\}$, i.e. the boolean values and the null value.
- the semantics of the type $\mathcal{G}$ is $\llbracket \mathcal{G} \rrbracket = \mathbb{G} \cup \{\omega\}$, i.e. the set of all graphs and the null value.
- for types of the form **Record**(ρ) the semantics $\llbracket \text{Record}(\rho) \rrbracket$ is the union of $\{\omega\}$ and the set of all records $[r]$ with $\text{domain}(r) = \text{domain}(\rho)$ and $\forall A \in \text{domain}(r) : r(A) \in \llbracket \rho(A) \rrbracket$.
- for types of the form **Bag**(T) the semantics $\llbracket \text{Bag}(T) \rrbracket$ is the union of $\{\omega\}$ and the set of all finite bags over $\llbracket T \rrbracket$.

We define $\mathbb{T}$ as the set of all types. If $T \in \mathbb{T}$ and $C \in \mathbb{C}$, then we say that the RDF value C has type T , written as $C : T$, if and only if $C \in \llbracket T \rrbracket$. Thus, a type that matches the value in Example 3.3.1 is the type

$$\text{Record}(\langle A : \mathcal{L}, B : \text{Bag}(\mathcal{U}), C : \text{Bag}(\text{Record}(\langle D : \mathcal{G}, E : \mathcal{L} \rangle)) \rangle)$$

Note that we intentionally don't say that "this is *the* type", as the presence of a *null* value means that the value of Example 3.3.1 matches more than one type.

Remark 3.3.7. From now on, if we talk about a value, we will mean a value for which a type is defined. As a type $\mathbf{Bag}(T)$ is only defined for bags of elements with type T , this means that a bag can only contain values of type T . Assuming that $r_1 = \langle A : l_2, B : l_3 \rangle$ with $l_1, l_2, l_3 \in \mathbb{L}$ we can observe that r_1 and l_1 have the different types $\mathbf{Record}(\langle A : \mathcal{L}, B : \mathcal{L} \rangle)$ and $\mathcal{L}$, respectively. Thus there is no type defined for $\{\{r_1, l_1\}\}$ and therefore it is not a legal value.

3.4 RGL Syntax

This section defines the syntax for *semantics functions*, which transform RGL values, and for *typing functions*, which are used to perform static typechecking on RGL expressions. Then the syntax of atomic processors and of workflows is defined. Finally a typing function is given for workflows.

The RGL language allows the user to compose semantics functions to manipulate RGL values.

Definition 3.4.1 (Semantics function). A semantics function over a finite set $K \subset \mathbb{K}$ is a partial function $\mathbb{C}^K \rightarrow \mathbb{C}$. It maps a row over values to an output value.

To allow static typechecking of expressions in RGL, we introduce the concept of a typing function that explicitly defines the output type of a semantics function, given its input types.

Definition 3.4.2 (Typing function). A typing function over a finite set $K \subset \mathbb{K}$ is a partial function $\mathbb{T}^K \rightarrow \mathbb{T}$. It maps a row over types to an output type.

If there is just a mention of a semantics function or a typing function, it will always be over some finite $K \subset \mathbb{K}$ – even when K is not explicitly defined.

3.4.1 Function syntax

Assume $\rho = \langle k_1 : T_1, \dots, k_n : T_n \rangle$, that is, $\rho \in \mathbb{T}^K$ is a row over types. The semantics $\llbracket \rho \rrbracket$ of ρ is the set of all rows over values that match that type. Formally, $\llbracket \rho \rrbracket = \{ \langle k_1 : v_1, \dots, k_n : v_n \rangle \mid v_1 \in \llbracket T_1 \rrbracket \wedge \dots \wedge v_n \in \llbracket T_n \rrbracket \}$. The set $\mathbb{A}$ is the set of all atomic function definitions, which are defined as follows.

Definition 3.4.3 (Atomic function definition). An atomic function definition is a tuple (f, t) , where

- $f : \mathbb{C}^K \rightarrow \mathbb{C}$ is a semantics function over K
- $t : \mathbb{T}^K \rightarrow \mathbb{T}$ is a typing function over K
- $\forall \rho \in \text{domain}(t) : r \in \llbracket \rho \rrbracket \rightarrow f(r) \in \llbracket t(\rho) \rrbracket$.

Note that both f and t are partial functions as per definitions 3.4.1 and 3.4.2.

3.4.2 Workflow syntax

Given a set D (it only makes sense to let this be a set of function definitions, but let it be an abstract set for now), the syntax of a workflow graph is defined as follows.

Definition 3.4.4 (Workflow Graph). A Workflow Graph (or simply workflow) over D is a tuple $(I, o, P, R, M, F, \sigma)$ where

- $I \subseteq \mathbb{K}$ is a finite set of workflow input ports
- $o \notin P$ is the workflow output port and is not used in any other workflow
- P is a finite non-empty set of processors (also called the processor output ports)
- $R \subseteq \mathbb{K} \times P$ is a finite set of processor input ports
- $M \subseteq R$ is a set of iterating input ports
- $F \subseteq (I \cup P) \times (R \cup \{o\})$ is the flow relationship
- $\sigma : P \rightarrow D$ is a definition function that maps each processor to a function definition.

We say that that $\{o\} \cup I \cup P \cup R$ is the set of workflow nodes. P and $\{o\}$ (and thus R) are disjoint with the set of workflow nodes of every other workflow.

Definition 3.4.5 (Well-formedness). A workflow graph $W = (I, o, P, R, M, F, \sigma)$ over D is well-formed if all of the following properties hold:

1. For each port $b \in \{o\} \cup R$ there is exactly one edge $(a, b) \in F$. This states that every input port has exactly one input.
2. The directed graph (P, E) with nodes P and edges $E = \{(p_1, p_2) \mid \exists k : (p_1, (k, p_2)) \in F\}$ is acyclic. This states that there are no cycles in the dataflow.
3. if for some $p \in P$ it holds that $\sigma_p = (I', o', P', R', M', F', \sigma')$ is a workflow graph, then:
 - a) σ_p is a workflow over $D' \subseteq D \setminus W$. This prevents circular 'nesting' of workflows.
 - b) σ_p is well-formed.
 - c) $I' = \{k \mid (k, p) \in R\}$. This states that the input portnames of p match the input portnames of σ_p .
4. for every atomic processor $p \in P$ with $\sigma_p = (Q, t) \in \mathbb{A}$, it holds that $\text{domain}(t) \subseteq \mathbb{T}^{I_p}$ with $I_p = \{k \mid (k, p) \in R\}$. This states that the processor input ports have the same key names as the input-mapping of the typing function of σ_p .

It only makes sense to construct workflows over function definitions:

Definition 3.4.6 (Function definitions). The infinite set $\mathbb{D}$ of function definitions is inductively defined as follows:

- $\mathbb{A} \subset \mathbb{D}$. That is, every atomic function definition is a function definition.
- if $D \subseteq \mathbb{D}$ and W is a well-formed workflow over D , then W is a function definition.

The set of all workflow graphs over $\mathbb{D}$ can be denoted $\mathbb{W}(\mathbb{D})$. The set of all workflow graphs $\mathbb{W}$ is now defined as the smallest set for which $\mathbb{W}(\mathbb{D}) \subseteq \mathbb{W}$.

From now on, if there is a mention of a workflow, it is assumed to be a well-formed workflow over $\mathbb{D}$. Now let us define some extra terminology.

Definition 3.4.7 (Atomic and Workflow processors). *A processor $p \in P$ is an atomic processor if it uses an atomic function definition: that is, $\sigma(p) \in \mathbb{A}$. If $\sigma(p)$ is a workflow over $D \subseteq \mathbb{D}$, then p is a workflow processor.*

We will also use σ_p to denote $\sigma(p)$.

Definition 3.4.8 (Nested Workflow Graph). *A workflow over $D \subseteq \mathbb{A}$ is a Flat Workflow Graph, as its definition function only specifies atomic processors. A workflow over $D \subseteq \mathbb{D}$ with $\exists p \in P : \sigma_p \notin \mathbb{A}$ is a Nested Workflow Graph. Furthermore, a Nested Workflow Graph is said to reuse another workflow W if there is a nested processor $p \in P$ and $\sigma_p = W$, or if $\sigma_p = Y$ and Y is a workflow that reuses W . Thus, the reuse relation is transitive.*

3.4.3 Type inference system

This section will specify a mechanism to determine whether an atomic processor or a workflow is *well-typed*.

If $W = (I, o, P, R, M, F, \sigma) \in \mathbb{W}$ is a well-formed workflow over D , then a typing function $t : \mathbb{T}^I \rightarrow \mathbb{T}$ exists for W . It is defined as follows .

Assume that for every atomic function $(f, t) \in D$ and workflow $(I', o', P', R', M', F', \sigma') \in D$ there is a typing function known. Assume $\rho : I \rightarrow \mathbb{T}$ is some row over types. We want to evaluate $t(\rho)$ to see how W transforms the types. The function $\tau_\rho : o \cup I \cup P \cup R \rightarrow \mathbb{T}$ maps any port in the workflow to a type, as defined by the following rules:

1. For any port $a \in R \cup \{o\}$ there is one port $b \in I \cup P$ such that $(b, a) \in F$, and $\tau_\rho(a)$ is defined as $\tau_\rho(b)$.
2. For any port $p \in P$, there is a function definition σ_p with some typing function t_p . If $\sigma_p \in \mathbb{A}$ then t_p is directly given because σ_p has the form $(\cdot, t_p)$. If $\sigma_p \in \mathbb{W}$ then t_p can be derived by recursively applying these rules.
We define $I_p = \{k \mid (k, p) \in R\}$ and $\rho_p : I_p \rightarrow \mathbb{T}$ as $\rho_p(k) = \tau((k, p))$; that is, ρ_p is a row over the types that are input to processor p . Then $\tau_\rho(p)$, the output type of p , is defined as

$$\tau_\rho(p) = \begin{cases} t_p(\rho_p) & \text{if } \forall k : (k, p) \notin M \\ \mathbf{Bag}(t_p(\rho_p)) & \text{otherwise} \end{cases} \quad (3.3)$$

3. For any port $k \in I$, we define $\tau_\rho(k)$ as $\rho(k)$

Thus τ_p is defined in terms of ρ , and the typing function t is simply defined as the type of the output port: $t(\rho) = \tau_p(o)$.

Note that the recursion in rule 2 always ends, as workflows cannot reuse themselves as per definition 3.4.2 (element 3a). Furthermore, note that Rule 2 may leave τ_p and thus t undefined if the typing function t_p of a processor implementation does not define a type for the input type mapping ρ_p . In other words, if the types propagated from the input ports of W are not accepted by the typing function of some processor in W , then t is undefined for that input typing. Finally note that if $I = \emptyset$, then $\langle \rangle$ (the empty row) is the only element in the domain of t .

Definition 3.4.9 (Well-typedness of a Workflow). *A workflow $W = (I, o, P, R, M, F, \sigma)$ is well-typed if, and only if, the derived typing function $t : \mathbb{T}^I \rightarrow \mathbb{T}$ has a non-empty domain. That is, there is a row over types $m_i : I \rightarrow \mathbb{T}$ for which t is defined.*

Remark 3.4.10. *Note that for a workflow $W = (I, o, P, R, M, F, \sigma)$ with $I = \emptyset$ it can be decided whether W is well-typed, as follows. By definition, the typing function is then $t : \mathbb{T}^{\emptyset} \rightarrow \mathbb{T}$, thus $t : \{\langle \rangle\} \rightarrow \mathbb{T}$. Then the computation of $t(\langle \rangle)$ can be attempted. If it is defined, then $\langle \rangle \in \text{domain}(t)$ and thus W is well-typed. Otherwise, $\text{domain}(t) = \emptyset$ and W is not well-typed. If $I \neq \emptyset$ the situation is more difficult. If a row over types ρ is suggested as a yes-instance in the domain of t , it can be verified by computing $t(\rho)$. If this is defined, then W is well-typed. Otherwise, as there are infinitely many nested types, we have no way to decide whether W is well-typed. An efficient decider may exist, but I am not aware of it.*

3.5 RGL Semantics

This section will describe the semantics of RGL. We will first specify the semantics of atomic functions, and then the workflow semantics.

3.5.1 Atomic function definition semantics

Definition 3.5.1 (Domain of an atomic function definition). *The semantics $\llbracket (f, t) \rrbracket$ of any $(f, t) \in \mathbb{A}$ is f .*

Note that $f : \mathbb{C}^K \rightarrow \mathbb{C}$ maps rows over RGL values to RGL values; it does not map records (although the row over values that is input to f may contain records).

The specification of the core functions of RGL is given in section 3.6.

3.5.2 Workflow semantics

Assume a workflow $W = (I, o, P, R, M, F, \sigma)$ and $r : I \rightarrow \mathbb{C}$ a row over RGL values that we call the *workflow input row*. A function $\phi_r : o \cup I \cup P \cup R \rightarrow \mathbb{C}$ assigns a value to every port when the row r is input to the workflow, and is defined by the following rules (note the similarity with the function τ_p that was used to derive the typing function t of W):

1. For any port $a \in R \cup \{o\}$ there is one port $b \in I \cup P$ such that $(b, a) \in F$, and we define $\phi_r(a)$ as $\phi_r(b)$.

2. For any port $p \in P$, there is a function definition σ_p for the processor with $\llbracket \sigma_p \rrbracket = f_p$. We define the set of input-keys of p as $I_p = \{k \mid (k, p) \in R\}$ and a row $r_p : I_p \rightarrow \mathbb{C}$ (a row over RGL values that are input to processor p) as $r_p(k) = \phi_r((k, p))$. We will see that, in the specific case where $\forall k : (k, p) \notin M$ (that is, no input ports of p are marked for iteration), the semantics is defined as the transformation of the input row r_p by the function f_p . Thus, $\phi_r(p) = f_p(r_p)$. However, we will proceed to define the general case.

For any $k \in I_p$ with $(k, p) \in M$, we say that *processor p iterates over the port (k, p)* . Note that the RGL-value $r_p(k)$ has the type $\mathbf{Bag}(T)$ with some $T \in \mathbb{T}$; this follows from Definition 3.4.9 and in particular Equation 3.3. For those $k \in I_p$ where $(k, p) \notin M$, the RGL-value $r_p(k)$ can be of any type. So for every $k \in I_p$ we can define a bag B_k of values, over which the input port (k, p) will iterate, as follows:

$$B_k = \begin{cases} \{\{r_p(k)\}\} & \text{if } (k, p) \notin M \\ r_p(k) & \text{otherwise} \end{cases} \quad (3.4)$$

For each B_k , we now define a bag $\mathbf{B}_{\{k\}}$ of rows over values, each record having domain $\{k\}$:

$$\mathbf{B}_{\{k\}} = \{(\langle k : s \rangle, n) \mid (s, n) \in B_k\}$$

If we say that $I_p = \{k_1, \dots, k_n\}$, we define $\mathbf{B}_{I_p}$ as the Cartesian bag-product $\mathbf{B}_{\{k_1\}} \overline{\times} \dots \overline{\times} \mathbf{B}_{\{k_n\}}$ (recall Equation 3.2). This cross product of all inputs contains rows $m_p : I_p \rightarrow \mathbb{C}$ for which the function f_p is defined. So finally the general definition of the output $\phi_r(p)$ of processor p can be given:

$$\phi_r(p) = \begin{cases} f_p(r_p) & \text{if } \forall k : (k, p) \notin M \\ \{(f_p(m_p), c) \mid (m_p, c) \in \mathbf{B}_{I_p}\} & \text{otherwise} \end{cases} \quad (3.5)$$

Thus, the iteration is very much like the mapping operator of functional programming languages, although it can also express the Cartesian product operation by marking multiple input for iteration.

3. For any port $k \in I$, we define $\phi_r(k)$ as $r(k)$

Thus we defined ϕ_r in terms of r , and we conclude the definition of the semantics of W as follows:

Definition 3.5.2 (Semantics of a workflow). *The semantics of a workflow $W = (I, o, P, R, M, F, \sigma) \in \mathbb{W}$ is the partial mapping $\llbracket W \rrbracket = f : \mathbb{C}^I \rightarrow \mathbb{C}$ defined as $f(m) = \phi_r(o)$.*

Remark 3.5.3. *Note that the semantics $f = \llbracket W \rrbracket$ of a W is not defined for every possible input row $r : I \rightarrow \mathbb{C}$. If t is the typing function derived for W , then $\text{domain}(t)$ defines for which input types f is defined. The rule $\forall \rho \in \text{domain}(t) : r \in \llbracket \rho \rrbracket \rightarrow f(r) \in \llbracket t(\rho) \rrbracket$ is propagated from Definition 3.4.3 to definition 3.5.2. Thus, as an input type is mapped to an output type by t , an input row over values that conforms to that type can actually be transformed by the workflow semantics function f .*

3.6 RGL core function definitions

The RGL functions can be RGL *core* functions and RGL *custom* functions. This section will explain the RGL core function. These form a small set of functions that are assumed to be available in every RGL implementation. Note that many of these functions are polymorphic. We will write

$$\frac{\begin{array}{c} \text{input}(k_1) : T_1 \\ \dots \\ \text{input}(k_n) : T_n \end{array}}{f(\text{input}) : T_{n+1}}$$

to describe a typing function $t : \mathbb{T}^K \rightarrow \mathbb{T}$ that defines the typing for a function f . It defines t for every row ρ over types with $\text{domain}(\rho) = \{k_1, \dots, k_n\}$ as $t(\rho) = T_{n+1}$ if and only if for all $i \in \{1, \dots, n\}$ it holds that $\rho(k_i) = T_i$. Together with f , this defines the (f, t) tuple of the atomic function definition syntax (see definition 3.4.3).

The functions in RGL core are the following:

- *Const* generates constants
- *Cmp* compares values
- *Math* allows the addition, subtraction, division and multiplication of number literals.
- *Condition* is used for if-then-else constructs
- *RecordCreate* constructs records from existing values
- *RecordProject* projects a value from a record
- *BagSingleton* creates a 1-element bag from a value
- *BagUnion* creates the union of two bags
- *BagMath* allows aggregate functions over bags of literal numbers (currently only Sum).
- *BagFilter* filters a bag for those elements that meet a certain criterion
- *BagFlatten* merges all bags in a bag.
- *BagCategorize* categorizes the elements in a bag in separate bags that are contained in a record
- *BagGroup* groups a bag of records by given field(s). It returns a nested type where records with the same value for those fields are grouped together.
- *LocalSPARQL* binds a number of variables in a SPARQL query and executes it on one or more graphs given as input
- *RemoteSPARQL* binds a number of variables in a SPARQL query and executes it on a SPARQL endpoint

Note that more functions can be implemented using the RDF Gears Java API (see section 5.5).

3.6.1 Basic functions

Const creates constant values that are non-nested (i.e. that are not of the **Bag** or **Record** type). It can thus generate the boolean values *True* and *False*, URI's, Literals and the null value ω . It is defined for some constant value v with the following typing function:

$$\frac{v : T}{\text{Const}_v(\langle \rangle) : T}$$

and semantics $\text{Const}_v(\langle \rangle) = v$.

Cmp compares non-nested values. For any comparator, the function typing is

$$\frac{\begin{array}{l} \text{input}(a) : \mathcal{V} \\ \text{input}(b) : \mathcal{V} \end{array}}{\text{Cmp}_*(\text{input}) : \mathcal{B}} \quad \frac{\begin{array}{l} \text{input}(a) : \mathcal{B} \\ \text{input}(b) : \mathcal{B} \end{array}}{\text{Cmp}_*(\text{input}) : \mathcal{B}}$$

The various comparators and their semantics are:

- $\text{Cmp}_{<}(\text{input}) = \begin{cases} \text{True} & \text{if } \text{input}(a) < \text{input}(b) \\ \text{False} & \text{otherwise} \end{cases}$
- $\text{Cmp}_{\leq}(\text{input}) = \begin{cases} \text{True} & \text{if } \text{input}(a) \leq \text{input}(b) \\ \text{False} & \text{otherwise} \end{cases}$
- $\text{Cmp}_{>}(\text{input}) = \begin{cases} \text{True} & \text{if } \text{input}(a) > \text{input}(b) \\ \text{False} & \text{otherwise} \end{cases}$
- $\text{Cmp}_{\geq}(\text{input}) = \begin{cases} \text{True} & \text{if } \text{input}(a) \geq \text{input}(b) \\ \text{False} & \text{otherwise} \end{cases}$
- $\text{Cmp}_{=}(\text{input}) = \begin{cases} \text{True} & \text{if } \text{input}(a) = \text{input}(b) \\ \text{False} & \text{otherwise} \end{cases}$
- $\text{Cmp}_{\neq}(\text{input}) = \begin{cases} \text{True} & \text{if } \text{input}(a) \neq \text{input}(b) \\ \text{False} & \text{otherwise} \end{cases}$

The ordering is defined only between values of the same type. It holds that

- all null values are equal
- null values are smaller than non-null values

For RDF values, the ordering is further defined by the SPARQL definition². For RGL Boolean values, $\text{True} > \text{False}$.

²<http://www.w3.org/TR/sparql11-query/#modOrderBy>

Math is defined for one of the operators $+$, $-$, $\times$, $\div$. It allows to perform mathematical operators on number literals, yielding a new number. Its typing function is

$$\frac{\begin{array}{c} input(a) : \mathcal{V} \\ input(b) : \mathcal{V} \end{array}}{Math_*(input) : \mathcal{V}}$$

The semantics is trivial:

- $Math_+(input) = input(a) + input(b)$
- $Math_-(input) = input(a) - input(b)$
- $Math_\times(input) = input(a) \times input(b)$
- $Math_\div(input) = input(a) \div input(b)$

Note that these semantics are defined only for literals that can be interpreted as a number type. The resulting literal is of the type $xsd : boolean$. It is thus not possible to guarantee computability of this operator by static typechecking.

Condition is used to select one of two inputs depending on the truth value of another value. It has a function typing t defined as³

$$\frac{\begin{array}{c} input("if") : \mathcal{B} \\ input("then") : T \\ input("else") : T \end{array}}{Condition(input) : T}$$

The semantics is defined as

$$Condition(input) = \begin{cases} input("then") & \text{if } input("if") = True \\ input("else") & \text{if } input("if") = False \\ \omega & \text{if } input("if") = \omega \end{cases}$$

3.6.2 Record functions

RecordCreate forms a record from values. It is defined for a set of inputs $k_1, \dots, k_n$. The typing function t is defined for input type row ρ as

$$\frac{\begin{array}{c} input(k_1) : T_1 \\ \dots \\ input(k_n) : T_n \end{array}}{RecordCreate(input) : \mathbf{Record}(\langle k_1 : T_1, \dots, k_n : T_n \rangle)}$$

³The implementation of RGL is a bit more liberal in that it supports subtyping for this operator, although we did not formally define this notion here. It means that the implementation can accept a record type input in the *else* port that has more fields than that input in the *if* port (or vice versa). The output type will be the common supertype.

The semantics is defined as

$$\text{RecordCreate}(\text{input}) = [k_1 : \text{input}(k_1), \dots, k_n : \text{input}(k_n)]$$

RecordProject takes a value from a record. It is defined for a certain field name k . The typing function t is defined for input type row ρ as

$$\frac{\text{input}(\text{"record"}) : \mathbf{Record}(\rho') \quad k \in \text{domain}(\rho')}{\text{RecordProject}_k(\text{input}) : \rho'(k)}$$

If $\text{input}(\text{"record"}) = [r]$ (remember r is a row over values) then the semantics is defined as

$$\text{RecordProject}_k(\text{input}) = r(k)$$

3.6.3 Bag functions

BagSingleton is used to created a bag from a value. The function typing t is defined for input type row ρ as

$$\frac{\text{input}(\text{"value"}) : T}{\text{BagSingleton}(\text{input}) : \mathbf{Bag}(T)}$$

The semantics is defined as

$$\text{BagSingleton}(\text{input}) = \{\{\text{input}(\text{"value"})\}\}$$

BagUnion merges two bags into one new bag. The function typing t is defined for input type row ρ as

$$\frac{\begin{array}{l} \text{input}(\text{"bag1"}) : \mathbf{Bag}(T_1) \\ \text{input}(\text{"bag2"}) : \mathbf{Bag}(T_2) \\ T_1 = T_2 \end{array}}{\text{BagUnion}(\text{input}) : \mathbf{Bag}(T_1)} \quad \frac{\begin{array}{l} \text{input}(\text{"bag1"}) : \mathcal{G} \\ \text{input}(\text{"bag2"}) : \mathcal{G} \end{array}}{\text{BagUnion}(\text{input}) : \mathcal{G}}$$

The semantics are defined as

$$\text{BagUnion}(\text{input}) = \text{input}(\text{"bag1"}) \cup \text{input}(\text{"bag2"})$$

Note how the $\cup$ operator is defined for graphs by the fact that graphs are sets of triples.

BagMath is defined for some aggregation function (currently only the summation function Σ). Its typing function is

$$\frac{\text{input}(\text{"numberbag"}) : \mathbf{Bag}(\mathcal{L})}{\text{BagMath}_\Sigma(\text{input}) : \mathcal{L}}$$

The semantics of the summation function is that all values in the bag will be added. The constraint on the input types, and particularly the constraint of statically type-checking these inputs, are the same as for the *Math* function defined above.

BagFilter filters a bag for elements that match a certain criterion. It is defined for a certain filter function $f : \mathbb{C}^{\{k\}} \rightarrow \mathbb{B}$ that takes one named argument (the name k does not matter) and returns *True* or *False*.

The typing function is

$$\frac{\text{input}(\text{"bag"}) : \mathbf{Bag}(T)}{\text{BagFilter}_f(\text{input}) : \mathbf{Bag}(T)}$$

If $\text{input}(\text{"bag"}) = B_{in} : \mathbb{C} \rightarrow \mathbb{N}$ (remember we defined bags as functions in section 3.2), then the semantics of $\text{BagFilter}_f(\text{input})$ is defined as a bag $B_{out} : \mathbb{C} \rightarrow \mathbb{N}$ with

$$B_{out}(e) = \begin{cases} B_{in}(e) & \text{if } f(e) = \text{True} \\ 0 & \text{otherwise} \end{cases}$$

BagFlatten takes a bag of bags and returns the union of all contained bags. The typing function is

$$\frac{\text{input}(\text{"bag"}) : \mathbf{Bag}(\mathbf{Bag}(T))}{\text{BagFlatten}(\text{input}) : \mathbf{Bag}(T)}$$

If $\text{input}(\text{"bag"}) = \{\{B_1, \dots, B_n\}\}$, then the semantics of $\text{BagFlatten}(\text{input})$ is defined as a $B_1 \cup \dots \cup B_n$.

BagCategorize is defined for a certain categorization function $f : \mathbb{C}^{\{k\}} \rightarrow \mathbb{V}$ with any k and a finite set of field names $k_1, \dots, k_n$. The categorization function assigns a Literal to an input. The BagCategorize function uses this categorization function to categorize every element $e : T$ of the input bag with type $\mathbf{Bag}(T)$ by using $f(e)$ as a record-fieldname.

The typing function is

$$\frac{\text{input}(\text{"bag"}) : \mathbf{Bag}(T)}{\text{BagCategorize}_f^{\{k_1, \dots, k_n\}}(\text{input}) : \mathbf{Record}(\langle k_1 : \mathbf{Bag}(T), \dots, k_n : \mathbf{Bag}(T) \rangle)}$$

Assume $\text{input}(\text{"bag"}) = B_{in}$. Then the semantics of $\text{BagCategorize}_f^{\{k_1, \dots, k_n\}}(\text{input})$ is defined as $[k_1 : B_1, \dots, k_n : B_n]$ with B_i defined for $1 \leq i \leq n$ as

$$B_i(e) = \begin{cases} B_{in}(e) & \text{if } \text{tofield}(f(e)) = k_i \\ 0 & \text{otherwise} \end{cases}$$

Here the function $\text{tofield} : \mathbb{L} \rightarrow \mathbb{K}$ converts a Literal value to a field name.

BagGroup is defined for a set of grouping field names $K = \{k_1, \dots, k_m\} \subset \mathbb{K}$. The typing function t is defined for a row over input types ρ as

$$\frac{\text{input}(\text{"bag"}) : \mathbf{Bag}(\mathbf{Record}(\langle k_1 : \mathcal{V}, \dots, k_m : \mathcal{V}, \dots, k_n : \mathcal{V} \rangle))}{\text{BagGroup}^K(\text{input}) : \mathbf{Bag}(\mathbf{Record}(k_1 : \mathcal{V}, \dots, k_m : \mathcal{V}, \text{"group"} : \langle \mathbf{Bag}(\mathbf{Record}(\langle k_1 : \mathcal{V}, \dots, k_m : \mathcal{V}, \dots, k_n : \mathcal{V} \rangle)) \rangle))}$$

Grouping creates a bag of records of type $\mathbf{Record}(k_1 : \mathcal{V}, \dots, k_m : \mathcal{V}, \text{"group"} : T_{\text{group}})$. Every such record has a unique combination of values for K . The "group" field contains all the records from the input bag that have the those values for the fields in K .

Assume $\text{input}(\text{"bag"}) = B_{in}$. Then $\text{BagGroup}^K(\text{input}) = B_{out} : \mathbb{C} \rightarrow \mathbb{N}$ is a bag that we will define (as a function) shortly. Assume a set of values $V = \{v_1, \dots, v_m\} \subset \mathbb{V}$. For this these values a bag B_V is defined as a set of $(\text{element}, \text{frequency})$ tuples as

$$B_V = \{ ([r], n) \mid ([r], n) \in B_{in} \wedge \forall i \in \{1, \dots, m\} : r(k_i) = v_i \}$$

Thus, the bag B_V has type $\mathbf{Bag}(\mathbf{Record}(\langle k_1 : \mathcal{V}, \dots, k_m : \mathcal{V}, \dots, k_n : \mathcal{V} \rangle))$ and contains all records with identical values for the field in K . The output bag B_{out} is now defined for any row over values $r_V = \langle k_1 : v_1, \dots, k_m : v_m, \text{"group"} : B_V \rangle$ with some $V = \{v_1, \dots, v_m\} \subset \mathbb{V}$ and B_V defined as above, as

$$B_{out}([r_V]) = \begin{cases} 1 & \text{if } \forall i \in \{1, \dots, m\} : r_V(k_i) = v_i \\ 0 & \text{otherwise} \end{cases}$$

Note that grouping is undone by mapping it with the $\text{RecordProject}_{\text{group}}$ function (applying a projection of the `group` field that iterates) and than flattening the bag⁴.

3.6.4 SPARQL functions

RGL supports the use of SPARQL to define atomic functions over input rows. We use $\mathbb{Q}_{\text{SELECT}}$ to denote the infinite set of SPARQL SELECT queries and $\mathbb{Q}_{\text{CONSTRUCT}}$ to denote the infinite set of SPARQL CONSTRUCT queries.

LocalSPARQL is defined for a tuple (q, B) , where

- $q \in \mathbb{Q}_{\text{SELECT}} \cup \mathbb{Q}_{\text{CONSTRUCT}}$ is a SPARQL query
- $B \subset \mathbb{K}$ is a set of field names that can be matched with a set of free variables names in q

⁴Also note that it may be more elegant to modify the typing and semantics of the Bag Group function to one where it produces simply a $\mathbf{Bag}(\mathbf{Bag}(\mathbf{Record}(\dots)))$ type (thus not providing the fields $k_1, \dots, k_m$ in a record that contains the inner bag). It can then be undone with just the Bag Flatten operation. The fields $k_1, \dots, k_m$ can be fetched from an arbitrary record in the inner bag, as all elements in one inner bag contain the same values.

The typing function is given by

$$\frac{\begin{array}{c} q \in Q_{SELECT} \\ input(b_1) : \mathcal{V} \vee input(b_1) : \mathcal{G} \\ \dots \\ input(b_m) : \mathcal{V} \vee input(b_m) : \mathcal{G} \\ S_q = \{s_1, \dots, s_n\} \end{array}}{LocalSPARQL_{q,B}(input) : \mathbf{Bag}(\mathbf{Record}(\langle s_1 : \mathcal{V}, \dots, s_n : \mathcal{V} \rangle))}$$

and

$$\frac{\begin{array}{c} q \in Q_{CONSTRUCT} \\ input(b_1) : \mathcal{V} \vee input(b_1) : \mathcal{G} \\ \dots \\ input(b_m) : \mathcal{V} \vee input(b_m) : \mathcal{G} \\ S_q = \{s_1, \dots, s_n\} \end{array}}{LocalSPARQL_{q,B}(input) : \mathcal{G}}$$

The semantics is as follows. The given SPARQL query q implies a query q' that is obtained by replacing, for every $b \in B$, all occurrences of a free variable named b in q by the value $input(b)$. Some injective function $gid : \mathbf{G} \rightarrow \mathbb{U}$ with $\mathbf{G} = \{g | g \in range(input) \wedge g : \mathcal{G}\}$ is defined that assigns a URI id to all graphs in the input row. If $input(b) : \mathcal{G}$ then $input(b) = G$ is a graph and the variable b in q is replaced by $gid(G)$.

The semantics $LocalSPARQL_{q,B}(input)$ is defined as the result of the SPARQL query q' over the named graphs defined by the gid function, using the SPARQL semantics defined by [20].

RemoteSPARQL is defined for a tuple (q, B, e) , where

- $q \in Q_{SELECT} \cup Q_{CONSTRUCT}$ is a SPARQL query
- $B \subset \mathbb{K}$ is a set of field names that can be matched with a set of free variables names in q
- e is a URI identifying a SPARQL endpoint

The typing function is given by

$$\frac{\begin{array}{c} q \in Q_{SELECT} \\ \rho(b_1) : \mathcal{V} \\ \dots \\ \rho(b_m) : \mathcal{V} \\ S_q = \{s_1, \dots, s_n\} \end{array}}{RemoteSPARQL_{q,B,e}(input) : \mathbf{Bag}(\mathbf{Record}(\langle s_1 : \mathcal{V}, \dots, s_n : \mathcal{V} \rangle))}$$

and

$$\frac{\begin{array}{c} q \in Q_{CONSTRUCT} \\ \rho(b_1) : \mathcal{V} \\ \dots \\ \rho(b_m) : \mathcal{V} \end{array}}{RemoteSPARQL_{q,B,e}(input) : \mathcal{G}}$$

As in the *LocalSparql* function, the query q implies a query q' that is obtained by replacing, for every $b \in B$, all occurrences of a free variable named b in q by the value $input(b)$.

The semantics of $RemoteSPARQL_{q,B,e}(input)$ is defined as the result of executing q' on the endpoint.

If $input(b)$ is a graph for some $b \in B$ than the behavior is not defined, as shown in the typing definition, as the locally created URI's are meaningless to the endpoint.

3.7 Evaluation

We evaluate the RGL language in a few ways. First the design rationale is discussed, arguing how alternative design choices would have yielded a language with less desirable properties. Then the expressivity of the language is discussed by relating it to the Nested Relational Calculus. The usefulness of the language is evaluated using the criteria defined earlier. Finally the optimizability of RGL is evaluated.

Strictly, this evaluation applies to the language itself and not to the RDF Gears framework as a whole. The extent to which these properties are expressed in the framework depends on the implementation of the user interface and the RGL engine, which will be evaluated in consequent chapters.

3.7.1 Design rationale

The design of the RGL language was a lengthy process with many design choices. The design rationale will be briefly clarified now.

Taverna inspired the decision to create a syntax similar to that of scientific workflow systems. This allows for more flexible, modular composition of transformation functions than (for example) with Silk. The possibility to nest workflows stimulates abstraction and can facilitate reuse of existing workflows. The language is kept purely functional and declarative in order to allow more straightforward application of query optimizations techniques that are known from database theory.

Furthermore a basic idea was to keep the data model close to that of RDF. For a reflection on the initial attempt to create a language with RDF graphs as the only data structure, see Appendix A. The adoption of the hybrid of NNRC with RDF datatypes as first-class citizens, instead of simulating them with RDF, provides a number of advantages. The hybrid of NNRC and RDF offers a more abstract representation of complex values (e.g. sets of records) than when such values would be simulated in RDF. This relieves the user from having to deal with such simulation details, and the NRC data structures and operations are shown to provide a query language that is

powerful yet easy to understand. Another advantage of this abstraction is that it gives implementations more flexibility and options in how and when it generates and stores the intermediate data (e.g. implementation choices like using pointers or id's, etc). Furthermore, with NNRC datastructures as first-class citizen, translating traditional NRC query optimization techniques [55] to RGL is likely to be easier (see Section 3.7.4).

The null value ω was included to provide a mechanism to deal with incomplete data and other errors without interrupting the dataflow. The idea is that a workflow will have to deal with incomplete or unavailable data frequently (e.g. when a SPARQL query using the OPTIONAL keyword has unbound variables, or when a query fails altogether). In such cases returning a null value would be more correct than assuming a default value. Also it is less disruptive than having the workflow execution fail altogether. Note that ω can be of any type and is thus not equivalent to the $()$ variable in NRC, which has the NRC *unit* type.

Wong [55] shows that adding a boolean type does not add any expressive power to the language. It was added to RGL for convenience.

The Bag Group operation was added to RGL to reduce the impedance mismatch [14] introduced by the flat format of the SPARQL SELECT results. The list of records returned by a SELECT query reduces the graph-based nature of RDF to an unnested relational model. Unfortunately, CONSTRUCT queries are no alternative as these have other practical disadvantages: the graph resulting from a CONSTRUCT query must again be queried with SELECT. Executing many small SPARQL queries is expensive in current implementations, and query paging (section 5.4.3) cannot be implemented for CONSTRUCT. The Bag Group operation makes it possible to query a dataset with a SELECT query and group records by one field to reconstruct a nested model that can be efficiently implemented and traversed (see section 5.4.1).

The mechanism of pre-binding SPARQL query variables with the function inputs was devised to make the SPARQL query mechanism more flexible. It allows the user to easily build a SPARQL query using intermediate results. Thanks to this powerful concept RGL can express constructs of federated query evaluation plans, such as semi-joins [40, 32]. Although it cannot be claimed that RGL will ever be able to compete with the performance of dedicated query federation systems, this may be very useful functionality in ad-hoc implementations of data integration algorithms that form part of a larger data transformation process. By extending the semantics of RemoteSPARQL to allow a *bag* of RDF values to be bound to a variable (instead of a *single* literal or URI), RGL can possibly even be integrated with the SPARQL federation extensions [39] in the future.

3.7.2 Expressivity

This formal account of RGL's expressivity is mainly of theoretical interest. It should be considered next to the more practical evaluation of the expressivity that follows from the design rationale and the examples shown throughout this thesis.

RGL directly adopted the datamodel of NNRC by Bussche and Vansummeren [10], but adopted bag-semantics instead of set-semantics for collections. NNRC is equivalent to the Nested Relational Calculus (NRC), of which the expressivity is researched by Wong [55] and Buneman et al. [9]. Wong also introduced the Nested Relational

Algebra (NRA) and proved that NRC and NRA are inter-definable, that is, that for every expression in one language an equivalent expression can be formulated in the other. The RGL core functions directly implement all core expressions of NRA and the acyclic directed graph syntax of RGL subsumes the tree-based expression syntax of NRA. Therefore RGL can simulate all expressions in NRA and thus also the languages NRC and NNRC. We thus conclude that RGL has at least the expressive power of these languages.

RGL can express what SPARQL CONSTRUCT and SELECT queries can express (this is trivial, as these SPARQL queries can directly be used in the LocalSPARQL function). SPARQL 1.1 provides a useful interface to RDF datasets and also allows operations not expressible in NRC, such as string operations (regular-expressions, substrings, etcetera) and the calculation of the maximum values.

There are, however, constructs that NRC nor SPARQL can express. Wong shows how NRC can be extended with extra operators to increase the expressivity. He adds the equality operator, creating the language $\mathcal{NRC}(=)$, for which he proves that the resulting language can be used to calculate constructs like set intersection, union and difference ([55] section 2.4). Wong furthermore discusses extra expressions such the mathematical operators and the summation function, creating the language $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, -, \cdot, \div, \Sigma, =)$ with $\mathbb{B}$ the boolean values, $\mathbb{Q}$ the set of rational numbers. He then shows how these operators can be combined to express aggregate functions like counting the number of elements in a set or bag, calculating the average value, and even calculating the variance ([55] section 3.2). These mathematical operators are all supported by RGL core and thus can be used to express these aggregate functions. Thus, RGL can simulate all expressions of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, -, \cdot, \div, \Sigma, =)$.

To express algorithms for which the set of operators is still insufficient, RGL can further be extended with custom operators. First of all, this is useful to increase the expressivity of RGL. Second of all, it can sometimes improve the usability, because the expression of some operations is cumbersome in RGL (e.g. because it requires complex simulations in NRC, or because it requires simulating complex nested collections in RDF and then using SPARQL to apply a regular expression). In the example of section 3.1 the custom operator *FetchRDF* was used, and section 6.3 will use an operator that calculates the Jaro string-similarity of two given strings. Custom functions can also be implemented to query relational databases, parse XML documents into nested collections, calculate aggregate functions, etcetera.

As arbitrary custom functions can be added to RGL, in fact any algorithm can eventually be implemented in this extended language (even when this needs to be done in a single RGL custom function). It is obvious that RGL may not be the right choice if the RGL datastructures and core functions do not allow modular implementation of the algorithm. For example, RGL may not be the right language to build a C compiler, although a C compiler could be wrapped in a custom RGL function. Such monolithic approaches will not be very useful. When custom functions can be added in modular components they may be a valuable addition to the language.

The fact that custom RGL functions can be added may leave the reader wondering about the purpose of formal analyzes of restricted languages like 'pure' RGL core. The answer is twofold. Firstly, a very expressive core language has the practical advantage that it allows to create complex transformations by combining simple predefined building blocks. By offering a toolkit of functions with the right level of abstraction

it is easier to use and learn. It also saves the user the work of implementing custom functions for simple tasks. The second reason for the importance of an expressive RGL core language is that it allows formal analysis of the semantics of workflows expressed by that core. This is useful for optimizations such as query rewriting: by changing the elements in an RGL expression such that the semantics is not affected, it is sometimes possible to allow for more easy evaluation. This is an important aspect in the efficient implementation of a query language like RGL. This will be discussed this in section 3.7.4.

3.7.3 Usefulness

The usefulness of the RDF Gears Language is now evaluated. It is concerned with the theoretic language, and not with the user interface and the engine implementation. The evaluation is done for each of the three criteria that were introduced in section 1.2.

The language is expressive enough to express custom data integration algorithms

The previous section has discussed the formal expressivity of RGL. It is based on a nested relational query language and is designed with extensibility in mind. As RGL also has these properties, it can express many kinds of algorithms. The interfaces offered to SPARQL endpoints provide the flexibility to create SPARQL queries based on intermediate values, which is a useful feature when aggregating data.

The arbitrary ways in which operators can be combined allows the quick implementation of mashups and integration algorithms. The first section of this chapter has shown how the language can be used to aggregate Linked Data from the web. More complex algorithms are expressible, as will be illustrated with the Silk-equivalent workflow presented in chapter 6. That chapter will also illustrate how an algorithm sometimes does not satisfy the requirements, and how the modular nature of a workflow language makes it possible to change such algorithms. RGL thus allows the user to express arbitrary algorithms and allows modification of algorithms to satisfy different requirements.

Both the Linked Data example of the beginning of this chapter as well as the Silk implementation discussed in chapter 6 require custom operator implementations. Respectively, these are the FetchRDF operator, which downloads Linked Data from the Web, and the TopScore operator, that selects some 'best' element of a bag using some criterion. This illustrates the necessity of an extensible language, or at least that the RGL-core operator collection alone is currently insufficient. With more experience in the practical use of RGL, it may be possible to find generalized, often used operators. These can be collected as some set of 'standard' operators or possibly be added to the set or 'core' operators.

A limitation of RGL is that, like RDF, the RGL data format is not well suited to express lists. Lists can be simulated in RDF, but this is inconvenient. Bags are unordered and thus intra-operator lists are not well-supported. RGL is not very suitable for dealing with ordered collections and algorithms requiring these may therefore not be well expressible. This is a design choice. Many algorithms do not require lists. Any defined order would take away useful freedom from a possible optimizer: the order must be taken into account by operators that merge bags, join bags of records and

iterate over multiple lists simultaneously with the Cartesian Bag-product. By giving the optimizer more freedom, it can use more liberal algorithms to implement such operations efficiently. If desired, the language semantics can be extended later by adding a List type (or replacing the Bag type with a List type). This would then be a backwards compatible change. In the meantime, ordered results can merely be used within operators that use an internal datastructure that maintains this order, such as a Java List.

The language provides a ‘right’ level of abstraction to make the expression of such algorithms convenient.

The previous section has shown the language is not too specific, as it still has high expressivity. The language is also not too generic, like a general purpose programming language, as it provides high level mechanisms. For example, a bag can be iterated simply by marking an input port. RDF can be fetched from the web with a custom operator that implements this functionality. The resulting data is automatically used to construct an RDF graph. So the user can query linked data without the need to deal with HTTP headers, catch HTTP exceptions, parse the RDF-XML, hard code or load a SPARQL query, execute it on the datasource, deal with query exceptions, and traverse the query results. Maintaining nested data structures for intermediate results need not be done with low level Java datastructures. Implemented workflows that have a clearly defined functionality can be reused within other workflows, only requiring the user to understand the interface.

We must note that the convenience of using this language also largely depends on the concrete syntax that is used and how this can be input to the computer system. An example is workflow nesting. Although this is a very intuitive way of reusing workflows and iterating over a series of operations, this may not be experienced as so convenient if the user interface does not allow a convenient way of visualizing this. As we will see in chapter 4, this is unfortunately the case in our GUI implementation – but this is not a language limitation.

The set of core operators from the NRA is originally intended to provide a formal expressivity and allows analysis of the language. For practical purposes, it misses some useful operations that must now be simulated. For example, it would be useful to add an operator that has n ports, whose input are merged in a bag of n elements. Currently, implementing this requires the generation of n singleton bags, and then using BagUnion operators to merge them pairwise. Such convenience operators can be added as syntactic sugar to this simulation mechanism, or implemented as operators carrying their own semantics.

A disadvantage that applies to the RGL syntax is that the parameters that are used by some functions, such as the function-criterion used to configure the BagFilter, cannot be configured by a super workflow. This would be a big advantage for the configurability and thus the reusability of a workflow that uses such a function. It may be possible to extend RGL with some syntactic means of configuring such a processor parameter in a higher level of the workflow reuse stack. This is outside the scope of this work.

The functionality is clearly defined and understandable.

The functionality is formally defined in this chapter to provide an unambiguous specification for language theoretic study by mathematicians and computer scientists. For mere mortals, the functionality can also be explained in less formal terms. This was done by means of an example in section 3.1. Experiences with Taverna show that its workflow paradigm is a good fit to the way users think about data transformation [37]. The similarity of RGL with the Taverna language suggests that RGL can also be understood by non-programmers.

It is true that the SPARQL construct of RDF Gears may be complex for some, as the SPARQL query language is nontrivial. Although I think that understanding basic SPARQL semantics is very useful for people working with Semantic Web data, not everybody will be comfortable to deal with these. It may be nice to provide custom functions that perform often used SPARQL query constructs, or to implement more advanced syntactic extensions of the language that can serve as a syntactic sugar.

3.7.4 Applicability of optimization techniques

The most important techniques that are currently implemented in the RDF Gears engine will be discussed in section 5.6. These are the pipelining mechanism in combination with the optimizer, the lazy evaluation and the Field Map indexing. The implementation shows that RGL is suitable for optimization techniques, thanks to its functional, statically typed nature.

Other optimization techniques exist, although they are not in the scope of this thesis. Let's provide some comments on the suitability of RGL for query optimization techniques, or more specifically, for query rewriting [11].

Wong describes optimization techniques for NRC and NRA, based on a list of axioms ([55] section 2.2). We can easily translate the NRA rewriting rules to RGL, but the equivalent rules in RGL must be generalized for two reasons. First of all, our syntax follows a graph structure whereas NRA uses a tree structure. Second of all, our syntax uses records (i.e. named tuples), whereas NRC uses 2-tuples only.

The syntax of NRA has not been explained in this document, but we will list some of the axioms and briefly clarify them. The $\sim$ symbol is used to denote that the workflow graph parts on the left and the right side of the $\sim$ symbol are semantically equivalent. The notation is not very formal, as this would be a notationally complex exercise and this is beyond our scope. The point of this section is to illustrate that such a formal rewriting system *can* be constructed for RGL.

- $\langle \pi_1 \circ f, \pi_2 \circ f \rangle = f$. This means that constructing a tuple from the projected elements of another tuple will yield the original tuple again. The equivalent rule for RGL is depicted in Figure 3.2a. The port f outputs a record with some set of fieldnames K ; in the example, $K = \{A, B, C\}$. The record create processor must create a record using exactly all projected fields of K . Then this structure is equivalent to connecting port f directly to the port g , the port reading from the record create processor. Note that other ports, not depicted in the picture, may read port f .
- $\pi_1 \circ \langle f, g \rangle = f$ and $\pi_2 \circ \langle f, g \rangle = g$. This means that constructing a tuple and then

projecting a given field will simply evaluate to the expression used to create that tuple field. The equivalent rule for RGL is depicted in Figure 3.2b. Note how it applies to any fieldname instead of any tuple position. Note that other ports, not depicted in the picture, may read any of the ports f or g . Also note that if g is not read by any other ports, the right version of this subgraph needs not evaluate g (but also note that the lazy evaluation implemented for RDF Gears, discussed in section 5.3.3, prevents g from being evaluated even without a query rewriting mechanism).

- $map(g \circ f) = map(g) \circ map(f)$. This means that iterating over the piped functions $g \circ f$ is equivalent to having function g iterate over the result of the iteration of f over the input. The equivalent rule for RGL is depicted in Figure 3.2c, assuming that no other ports read the output of the processors of functions f or g . Note that other ports, not depicted in the picture, may read port h .
- $map(f) \circ \eta = \eta \circ f$. This means creating a singleton set of a value and then iterating over that singleton with f is equivalent to applying f to the original input and then creating the singleton (not graphically illustrated here).
- $map(f) \circ \mu = \mu \circ map(map(f))$. This means that flattening a set and then mapping it with function f is equivalent to mapping it with the function $map(f)$ and then applying the flattening operation (not graphically illustrated here).

These axioms are further used to formulate a set of optimization rules ([55] chapter 6) for NRC. The NRA equivalent rules can be directly applied to RGL where they are used in a tree-structure (care must be taken to apply the rules on parts of a workflow that form a graph-structure). Wong shows this system rewrites any expression to a normal form in a finite number of steps. This rewriting system is for example very interesting for the RGL Bag-filter construct. Assume a filter uses a function with a selectivity of $x\%$. Wong shows how queries can be reordered such that the cost (with respect to the data size) of evaluation is reduced to $x\%$ of the original cost. This is done by reordering the query such that the filter is executed as soon as possible, thus eliminating unnecessary work on elements that will be filtered out later.

Finally let us note that interesting optimizations may be achieved by query optimizations that can reorder and combine SPARQL and NRC constructs. Just like the query rewriting mechanism for NRA, where a filter is moved as close to a generator as possible, this filter construct may in some cases be included in the SPARQL query. This may give room to the SPARQL engine to perform query optimizations, and – in the case of remote SPARQL endpoints – will reduce the amount of network bandwidth consumed by data elements that will be removed by the RGL filter later.

The other way around, if SPARQL constructs are used in the middle of an RGL workflow, optimization rules may be discovered that move parts of the SPARQL query (e.g. the filter that was just pushed into it!) to an earlier part of the RGL workflow. In the case of a highly selective filter operator, for example, this may be very attractive because it reduces the intermediate result sizes sooner and thus prevents more unnecessary evaluations.

These rewriting possibilities that mix NRC and SPARQL are fundamental, and they are mentioned here because these possibilities are considered a feature of the

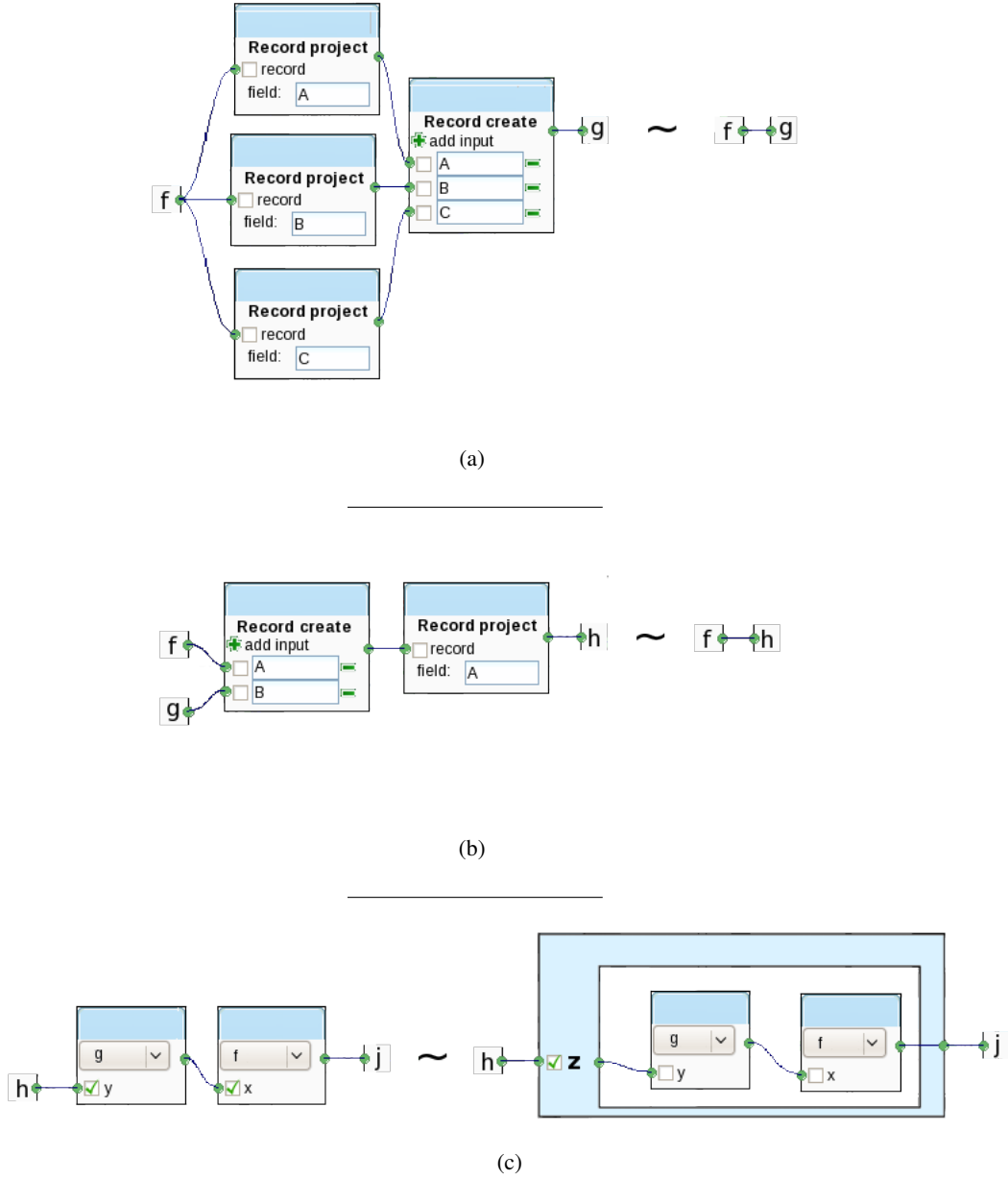


Figure 3.2: Some RGL query rewriting rules

RGL language. However, a proper rewriting system requires fundamental research on the relation between the SPARQL algebra and the Nested Relational Algebra, which is beyond the scope of this thesis.

Other optimizations techniques that are outside our scope but are likely to be implementable are dynamic techniques like cost-based optimization using query evaluation plans [12, 32] as well as implementations on the Map Reduce framework.

3.8 Conclusion

The RDF Gears Language (RGL) shows that it is possible to design a query and data transformation language for Semantic Web data. Section 3.1 presented a graphical, concrete syntax and thus provided an answer to research question 3 (an XML based concrete syntax will be presented in the next chapter).

RGL adopts as base data types the RDF elements Literal, URI's and graph. Furthermore it allows complex nested collections with Records and Bags that we know from the Named Nested Relational Calculus. Functions operating on these data structures can be flexibly combined to manipulate them. Custom functions can augment the expressivity of the core RGL language. Workflows can be nested not only to iterate over part of a larger workflow, but also to promote modularity and enable the reuse of workflows. The language is defined with a formal syntax and semantics, thus answering research question 1.

Section 3.7.2 showed that the RGL syntax and core functions can simulate the NRC language $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, -, \cdot, \div, \Sigma, =)$ by Wong. This language is shown to be equally expressive as other nested query languages and allows the creation and projection of records, collection operations like union, intersection, difference, Cartesian product of collections, mapping transformation and aggregate functions. These properties are inherited by RGL. Furthermore, RGL also allows fetching data from SPARQL endpoints with the RGL SPARQL operators and manipulating intermediate data structures with SPARQL. This makes RGL a highly expressive language that stays close to the RDF paradigm. Custom function can further augment the expressivity. This is a theoretic answer to research question 2.

The evaluation of RGL presents a design rationale that clarifies the most important of the many design choices that were made in the RGL design process.

The evaluation also assessed the usefulness of the language (section 3.7.3). Although the scope of the project does not allow a full study of the usefulness of RGL, there are strong suggestions that RGL is useful to implement new algorithms in Semantic Web data integration. First of all, this is suggested by the formal expressivity and the fact that it can be extended with custom operators. Secondly, the language offers a 'right' level of abstraction: the data model that is flexible and stays close to the RDF paradigm; operators that deal specifically with RDF datasets, such as the SPARQL functions, allows to quickly work with RDF data; the workflow paradigm allows modularization, and the functional nature allows the focussing on the conceptual data transformation and not on implementation details. These allow to quickly implement RDF data transformations in an intuitive way. Third of all, the workflow language is formally defined and the paradigm has been reported to be understandable by non-programmers dealing with data integration. (These properties are illustrated by the examples of section 3.1 and, as will be presented later, section 6.3). This way, it is shown that RGL meets the three criteria for usefulness: question 4 is answered.

Finally the evaluation discusses how the language can be optimized. It mainly focusses on NRC optimizations, and how these are applicable to RGL. It illustrates the suitability of RGL for optimization techniques, even though these are not currently implemented. It is a partial answer to research question 5. (Further answer to this question will be given in an evaluation of the actually implemented optimizations, presented in sections 5.6 and 6.5).

Chapter 4

Graphical User Interface

This chapter will present the RDF Gears User Interface (UI), a user interface prototype.

Section 4.1 introduces the rest of the workflow development environment with screenshots. In section 4.2 we present the XML format used to serialize the workflows. Section 4.3 provides a high-level architectural description.

4.1 User interface overview

The user interface prototype is a web application that can be hosted on a public web-server. Therefore users do not need to individually install the user interface, but can instead simply access it with their browser.

A screenshot of the GUI is shown in figure 4.1. The largest part of the workflow is consumed by the visualization of the workflow. The left panel shows the available functions in the core of RDF Gears, which can be dragged into the workflow. The top panel allows the user to save a workflow, give it a description, and to protect it with a password.

4.2 Produced XML format

The workflows created with the GUI are stored on the file system in the XML format. This format can be interpreted by the RGL engine implementation (discussed in Chapter 5). The files are stored on disk in the workflow directory named `rdfgears`. By default, the workflow directory resides in the operating system temporary directory, e.g. on Unix it is `/tmp/rdfgears/`. For every workflow a separate XML file is created, with the `.xml` extension. So if a workflow reuses other workflows, multiple files together form the full workflow specification. A workflow file is stored in the path `<workflow_directory>/<workflow_name>.xml`.

As an example the workflow `silkGroup/minimumScore` is depicted in Figure 4.2. It receives a record with name `input1` that is required to have a `score` field. If `input1.score ≥ 0.9` then the workflow returns `True`. It returns `False` otherwise.

The XML representation of this workflow must be stored in the file `/tmp/rdfgears/silkGroup/minimumScore.xml` (note how the slashes in the workflow name create a directory structure) and looks as follows:

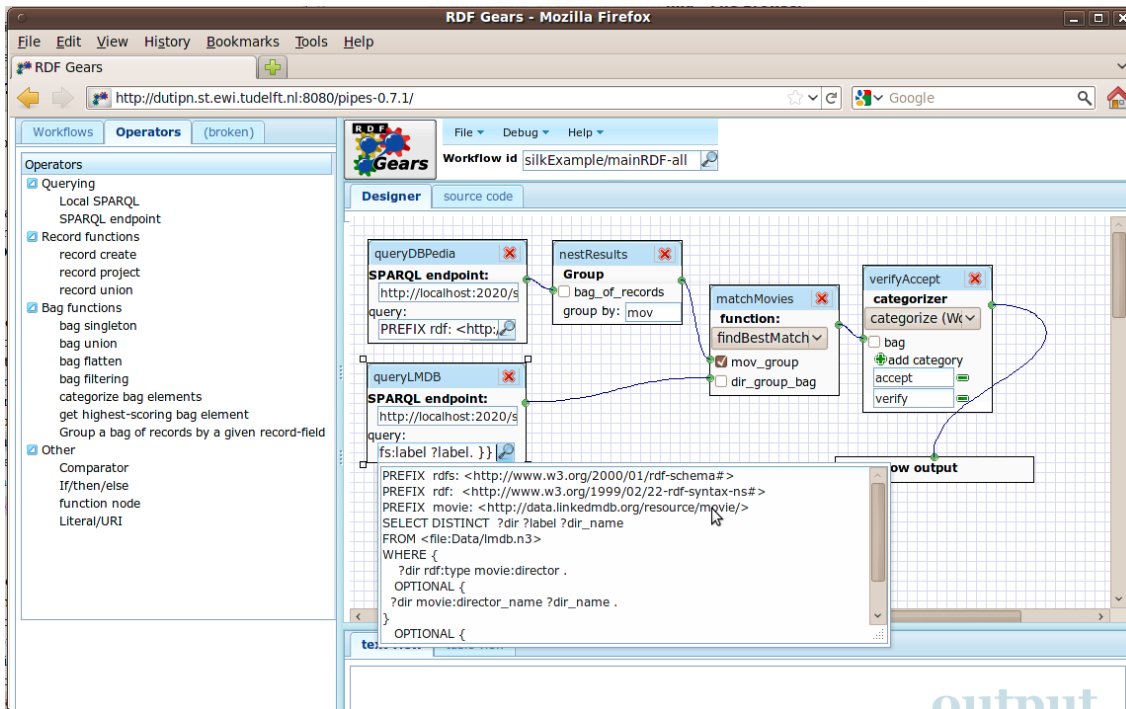


Figure 4.1: A screenshot of the RDF Gears GUI

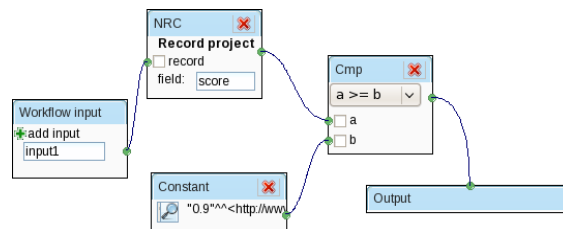


Figure 4.2: The workflow silkGroup/minimumScore

```
<?xml version="1.0" encoding="UTF-8"?>
<rdfgears>
  <metadata>
    <id>silkGroup/minimumScore</id>
    <description>Return true iff the score field of given records is
    0.9 or larger</description>
    <password/>
  </metadata>
  <workflow>
    <workflowInputList x="10" y="100">
      <workflowInputPort name="input1"/>
    </workflowInputList>
    <network output="node_276" x="352" y="182">
      <processor id="node_276" x="315" y="58">
        <function type="comparator">
          <config param="operator">OP_GREATER_EQUAL</config>
        </function>

```

```

        <inputPort iterate="false" name="a">
            <source processor="node_275"/>
        </inputPort>
        <inputPort iterate="false" name="b">
            <source processor="node_277"/>
        </inputPort>
    </processor>
    <processor id="node_275" x="140" y="14">
        <function type="record-project">
            <config param="projectField">score</config>
        </function>
        <inputPort iterate="false" name="record">
            <source workflowInputPort="input1"/>
        </inputPort>
    </processor>
    <processor id="node_277" x="144" y="170">
        <function type="constant">
            <config param="value">
                "0.9"^^<http://www.w3.org/2001/XMLSchema#double>;
            </config>
        </function>
    </processor>
</network>
</workflow>
</rdfgears>

```

If a user cannot access the filesystem of the webserver, the generated workflow XML can also be inspected and copied from the 'source code' tab of the workflow editing canvas.

4.3 Technical architecture

This section provides a high-level architectural description of the user interface. It mainly focuses on the relation between the user interface code and the libraries and frameworks it depends on.

The user interface is a Java web application that can be run in your favorite Java application server (e.g. Jetty is lightweight and simple).

The implementation of RDF Gears UI is based on the user interface implementation of DERI Pipes [36]. Although the RDF Gears UI is a fork that is now incompatible with the DERI Pipes codebase, it does largely inherit the architecture (see Figure 4.3). The UI is implemented in Java and built on top of the ZK Framework. It is an AJAX framework that allows the creation of web applications in Java without the need to write custom Javascript code. Although the current release of this Open Source framework is ZK 5.0, RDF Gears is still based on the older 3.6 version because this is what the codebase forked from DERI Pipes used. All tools in the user interface of RDF Gears are created with ZK, except for the workflow canvas.

The workflow canvas in the browser is drawn by the Javascript elements of the ZK framework and by the Draw2d library. Draw2d is a pure Javascript library that allows drawing and moving objects in a browser window. This functionality is used to draw the workflow ports and connections between them. The Javascript model is

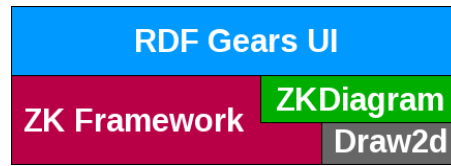


Figure 4.3: RDF Gears UI code and its relation to third party libraries

not directly manipulated by the RDF Gears code. Instead, RDF Gears UI uses ZK and ZKDiagram to manipulate the model in Javascript.

The ZKDiagram library provides AJAX functionality for Draw2d, by integrating it with the ZK framework. It was created by Danh Le Phuoc (DERI) for the implementation of DERI Pipes. It implements user interface elements like ports and connections both in Javascript and Java. The Javascript model uses the Draw2d API for visualization, and the Java model integrates with the ZK API to allow manipulation of the model on the server side. ZKDiagram transparently synchronizes the states between the two models.

Workflows can be executed on the server by clicking on the button with the workflow id, in the workflow list. This forwards the user to a HTTP URI specific for that workflow. In response, the web server launches the engine and evaluates the workflow, after which the results are returned to the client. This is a proof-of-concept of using RDF Gears to generate Linked Data on-the-fly. As the current link between the user interface and the execution engine is not optimized and execution may require a significant amount of resources, this will not work well for all workflows. Therefore it is recommended to run the engine on the client computer and to disable server side workflow execution, that is, disable execution with the graphical user interface.

4.4 Evaluation

The user interface implementation is a prototype that illustrates how a graphical syntax can be used to create workflows. It allows to model the workflows and edit them in a drag-and-drop fashion, similar to Taverna and DERI Pipes. The similarity to the syntax of Taverna suggests that such a graphical representation is useful for a wide audience, as it makes workflow specification intuitive and easy.

The visual workflow representation is easy to understand for people. For computers, the XML based serialization mechanism is relatively simple and works well. It is much cleaner than the DERI Pipes XML serialization, which nested the actual XML in CDATA text fields. The separation of RGL workflows in different documents allows reuse of workflows, but a disadvantage is that it is currently not trivial to collect all files that are necessary to execute a nested workflow. Another detail is that the workflow name mentioned in the XML must correspond to the XML documents file name, or the workflow will not be shown in the GUI workflow list. This is a redundancy that would ideally be removed to make manual renaming of workflows easier.

Unfortunately the DERI Pipes code uses a rather outdated technology stack (ZK 3.x). It works best on Mozilla Firefox – modern versions of other browsers do not work well. There are 2 known bugs in the system that are due to the used libraries. These

are also present in the DERI Pipes release. Firstly, edges are sometimes not properly removed when the user tries to delete them, or edges reappear when they have been deleted. A workaround is to save and reload the workflow. The bug is probably in code of the ZKDiagram and/or the Draw2d library. The second bug is that when editing the SPARQL query in the query operator multiple times, sometimes all newlines are removed. This issue must probably be fixed in the class `org.deripipes.ui.TextBandBox` or in one of its dependencies in the ZK library, but it may require JavaScript programming.

The advantage of the web application architecture is that interested researchers can experiment with the user interface by opening the publicly accessible web server in their browser, and therefore do not need to install any applications. Also, the publicly available workflows and password protection option inherited from DERI Pipes illustrates how a web application can be used for collaborative creation, demonstration and reuse of workflows.

Many minor things can still be improved on the interface. An ideal product would visualize the nested nature of workflow by showing reused workflows within the visualization of the reusing workflow. Also, it would have a feature to search for a workflow, real-time visual typechecking feedback, etcetera. So the prototype is not the be-all and end-all user interface, as this is beyond scope of this research project. However, the GUI does allow us to build workflows without writing XML documents, and it does provide a very concrete tool to present the concept and capabilities RDF Gears to the research community. Also, it can serve as a basis for future UI design. We thus conclude that it serves its purpose as a prototype well.

4.5 Conclusion

Inspired by graphical syntax of Taverna, the RDF Gears User Interface makes workflow editing easy. The GUI is considered a prototype to accompany RGL and the execution engine with a workable user interface and to illustrate the possibilities of such an environment. It is an easy to use, publicly accessible web application with which RGL workflows can be created, edited, inspected and deleted without the need to install any software other than Mozilla Firefox.

The XML format provides a clear and non-ambiguous serialization format for RGL workflows. This approach separates the implementation of the user interface from the execution engine. It is thus possible to execute a workflow without the need to install the user interface application. Also, it provides for a clean separation of concerns. This is attractive because it allows the mentioned improvements on the user interface without affecting the engine implementation that we will discuss in the next chapter. The XML format is another concrete syntax for the RGL language, next to the already presented graphical syntax. It completes the answer to research question 3.

Although the current implementation is useful, more features and usability improvements can be delivered in the future. With the current implementation RDF Gears can be used today. This is important because practical usage of the system will bring to light its strengths and weaknesses, allowing people to improve it.

Chapter 5

Implementation of the RDF Gears Engine

The RDF Gears Engine is an interpreter for RGL. This Chapter provides an insight in the implementation architecture of RDF Gears and in the optimizations that are implemented.

Section 5.1 presents the engine executable and its command line options. Section 5.2 discusses the engine architecture. It discusses the basic structure of values, functions and workflows and the typechecking and execution mechanism. The most important optimizations are discussed in section 5.3, among which are the pipelining architecture, the optimizer that determines where values must be kept in memory, and the lazy evaluation mechanism. Section 5.4 discusses the implementation of some of the RGL core functions. It shows how these fit in the architecture and how they automatically benefit from the optimizations. Section 5.5 briefly explains how custom functions can be implemented. Section 5.6 presents performance measurements to evaluate the optimizations by the pipelining optimizer and the Field Map Indexing.

```
[eric@dutipn] $ ./rdfgears --help
RDF Gears version 0.1
Use the --help option for usage info

The options available are:
  [--debug-level -d value] : The log4j debug level
                           (DEBUG/INFO/WARN/ERROR/OFF etc)
  [--disable-optimizer] : Disable the workflow optimizer
  [--execute -x] : Immediately execute, do not typecheck
  [--help] : show this help message
  [--outputformat value] : The format for the RGL output data
                           [xml|informal|none]. Default: xml
  [--typecheck-only -t] : Do not execute, only typecheck
  --workflow -w value : Path to the workflow to be executed
  [--workflow-path value] : List of ':' separated paths where the
                           (nested) workflows can be found. Default: ../workflows/
```

Figure 5.1: RDFGears engine options

5.1 Command line interface

The RDF Gears engine can be called with the `rdfgears` command. The command line options are shown in figure 5.1. As we can see, workflows can be typechecked and executed. The output can be formatted in different formats (XML and a more human readable format called 'informal').

The engine checks whether a workflow is well typed. As pointed out by remark 3.4.10, there currently is only a decider that determines whether workflows without input ports are well typed ($I = \emptyset$). As these are also the only workflows that can be evaluated, the engine can only typecheck such workflows. If a workflow is not well-typed, `rdfgears` provides a trace to the complaining processor so the user can inspect the issue. This is illustrated in figure 5.2. Note that the engine even correctly recognized that the user forgot an iteration marker, in this case. If the user marks a processor for iteration, the workflow is well-typed.

RDF Gears uses a configuration file `rdfgears.config`. This file is used to configure engine parameters. These are the default list of workflow paths; whether or not to use query paging (see section 5.4.3) for SPARQL endpoints, and the desired page size; whether or not paged queries should be aggressively loaded, thus detecting errors as early as possible; the number of retries that should be attempted when a remote SPARQL query fails; the time to pause between these SPARQL retries; the default debug level.

```
[eric@dutipn] $ ./rdfgears -w workflows/extractData/linkedmdb_dir_label
The workflow is not executable, as it did not pass the typechecking test:
I think you forgot an iteration marker somewhere!
Port 'record' received input of type Bag( Record(< v0:RDFValue,
b:RDFValue, >) ), but I require Record(< b:AnyType, >)
  in java-function nl.tudelft.rdfgears.rgl.function.core.RecordProject
    used by processor node_199
  in workflow extractData/linkedmdb_dir_label_createrecords
    used by processor node_230
  in workflow workflows/extractData/linkedmdb_dir_label
```

Figure 5.2: A typechecking error is detected by the engine

5.2 Basic architecture

The RDF Gears Engine is implemented in the Java package `nl.tudelft.rdfgears`. For brevity, we will omit this prefix when talking about sub-packages and classes within this packages (e.g. the class `nl.tudelft.rdfgears.engine.ValueFactory` is referred to as `engine.ValueFactory`). It uses the Jena framework with the ARQ libraries to manipulate and query RDF data.

5.2.1 RGL Values

The `RGLValue` class hierarchy stores the data that flows through a workflow. It is useful to understand the class hierarchy, especially when implementing (custom) RGL

functions. `RGLValue` and its subclasses are depicted in the class diagram of Figure 5.3. Now only the `DeterminedRGLValue` descendants are discussed; `LazyRGLValue` is discussed in section 5.3.3.

The classes with italic classnames are abstract. The only concrete classes shown are the `BagValue` implementations. Where the other implementations are omitted for conciseness, some `BagValue` implementations are shown because these are particularly interesting to illustrate the flexibility of the abstract hierarchical approach.

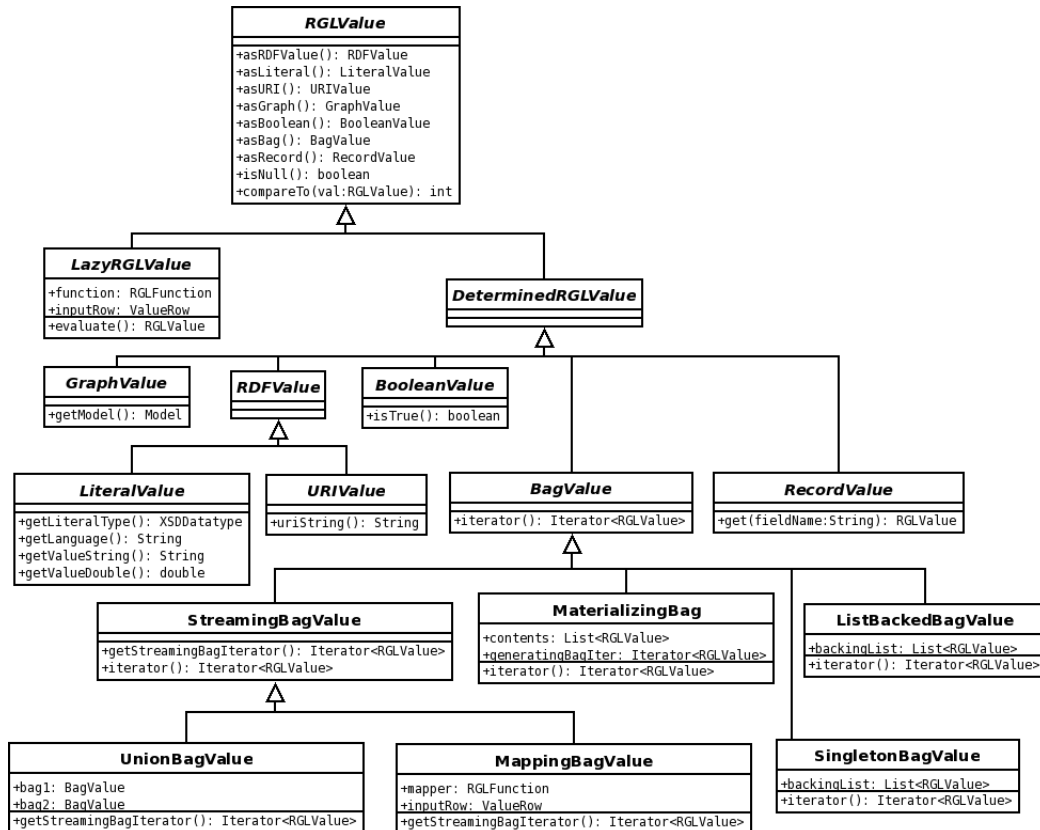


Figure 5.3: Class diagram of the RGL Value class hierarchy

The class `engine.ValueFactory` (figure 5.4) provides a number of static methods to instantiate values. It combines the Factory pattern with the delegation pattern so that a function implementation can easily call static `ValueFactory` methods. The `ValueFactory` class delegates the call to the equivalent dynamic method of the configured instance of the `ValueFactoryIface` class. This allows the factory implementation to be configured at runtime in a configuration file without the function implementation being bothered by this. Currently the `MemoryValueFactory` class implements the `ValueFactoryIface`. Other `ValueFactoryIface` implementations may create different `RGLValue` value implementations, e.g. a `DiskValueFactory` can create disk-based `RGL` values that work with a memory-based caching system.

The `LiteralValue` class represents RDF literals. In RDF a literal is either *plain* or *typed*. A typed literal is defined by a value and type URI. The string-encoded value is returned by the `getValueString()` call. The

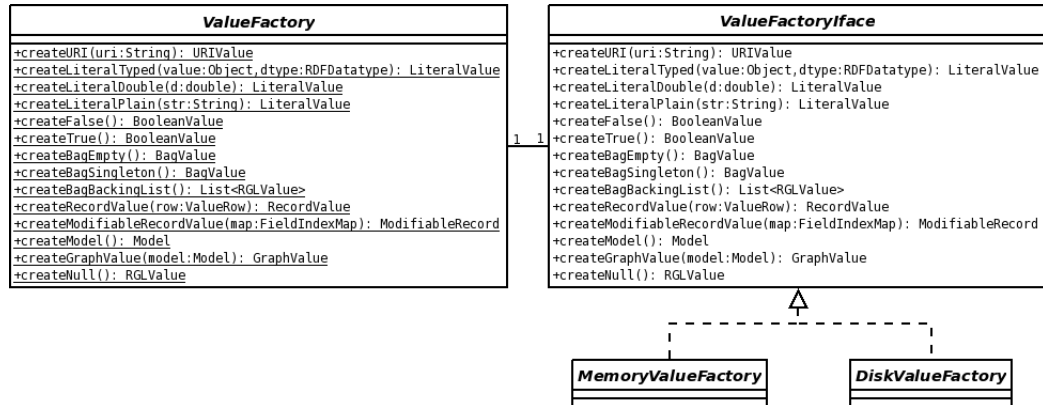


Figure 5.4: Class diagram of the RGL Value class hierarchy

internal representation may be not be String encoded for efficiency reasons. The call `ValueFactory.createLiteralDouble(0.2)` is for example delegated to `MemoryValueFactory#createLiteralDouble(double d)`, which creates a `LiteralValue` implementations with internal double representations. The `getValueDouble()` call returns this double. Other `LiteralValue` implementations may return null on the `getDoubleValue()` call if it cannot be converted to a double, or may need to string-parse the value representation to a double.

By implementing calls like `getValueInt()`, `getValueDate()` or `getValueDateTime()`, other Java objects or plain types may be returned. Note that typed Literal objects may also internally use such object/type to encode its value, which is more space efficient than string encoding.

A plain literal in RDF contains just a string, fetched with the `getString()` method. A language tag may be associated, returned by the `getLanguage()` method. If no such tag is set, it returns null (remember that a typed literal never has such a tag).

The `URIValue` is an abstract class whose implementations must simply return a string when `uriString()` is called. Therefore not many implementations are expected to arise. `ValueFactory.createURI()` is used to create an instance.

The `BooleanValue` classes only have an `isTrue()` method that returns the boolean value. The calls `ValueFactory.createTrue()` and `ValueFactory.createFalse()` return singleton instances of these booleans.

The `GraphValue` implementations are used to represent RDF Graphs. The call `ValueFactory.createGraph(model)` creates a graph using a Jena Model that can be accessed with the `getModel()` method. The model can be instantiated with the `ValueFactory.createModel()` call, which may create a disk-based or memory based model depending on the factory implementation. Other graph values may be implemented later, e.g. graph values that download their RDF data on-demand.

The `RecordValue` implementations provide a `get(String fieldName)` method that allows accessing the value under a certain `fieldName`. The method `ValueFactory.createModifiableRecord(FieldIndexMap map)` allows creation of `ModifiableRecord` values which also provide the `put(String fieldName, RGLValue value)` method that allows setting the record values. The `FieldIndexMap` is explained in section 5.3.4.

The `BagValue` is one of the most interesting RGL values as it has many different implementing subclasses. Every implementation must provide an `iterator()` method that returns a Java iterator over its RGL values. The iterator of the `ListBackedBagValue` simply iterates over an internal `List` of `RGLValues` (instantiated with `ValueFactory.createBagBackingList()` so the `ValueFactory` may create a disk-based list). Most RGL function implementations will not use a `ListBackedBagValue` but instead provide their own `BagValue` implementation. This allows pipelined calculation of a bags contents, either in a fire-and-forget style or with a caching mechanism. This depends on the circumstances and is explained in section 5.3.

5.2.2 RGL Functions

The formal, abstract syntax definition of an RGL function is given in section 3.4. The Java class hierarchy used for RGL function implementation is shown in figure 5.5. Note that a `ValueRow` is a row over values, and thus a mapping of fieldNames to `RGLValues`. A `ValueRow` is used to represent the named arguments for the function and behaves like a `Map<String, RGLValue>`.

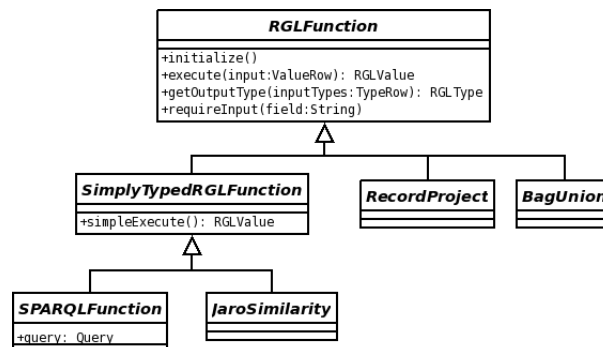


Figure 5.5: Class diagram of the RGL Function

The `RGLFunction` class has an `execute()` function, whose implementation defines the function behavior. It maps a `ValueRow` to a value. The function typing definition must define the domain and range by mapping a row over input types to an output type (see definition 3.4.2). If the output is always of the same type, the function can be implemented by extending the `SimpleTypedRGLFunction` class. If the function output type depends on the type of the input(s), this requires a slightly more complex function typing definition for which the `RGLFunction` must be implemented directly. The function `initialize(Map<String, String> config)` is used to perform configuration of the function behavior once, before the workflow execution. It is used, for example, to configure the projection fieldname of the `RecordProject` function, and the query string of the `SPARQLFunction`. Note that, like in these two examples, the values configured with the set with the `initialize` method may determine the typing properties of the processor.

5.2.3 Workflows and processors, execution and typechecking

Workflows and processors are used to model the workflow graph and execute it. The class `rgl.workflow.Workflow` is also an RGL function implementation, as shown in figure 5.5. A `Workflow` object models the workflow graph structure of definition 3.4.4 by connecting `Workflow` input ports and `Processor` objects. The `FunctionProcessor` instances contain an RGL function as attribute. Again, this may be a `Workflow` or an `AtomicRGLFunction`.

The typechecking mechanism is used to verify whether a workflow is well-typed. The typechecking is executed once, after all processor functions received their `initialize` call and before the actual workflow execution. The types are pulled from the end of the workflow by recursively calling `getOutputType(TypeRow inputTypes)` in a way similar to that of workflow execution, which is explained below. The `getOutputType` method of an RGL Function implementation can signal that a given row over types is not in the domain of the typing function by throwing a `FunctionTypingException`. This is caught by the wrapping workflow and a new exception is created with a reference to the original one. This creates the useful workflow stack trace of figure 5.2.

If the workflow is well-typed, it can be optimized by the optimizer that is discussed in section 5.3.2. After that, `Workflow#execute(ValueRow input)` is called to execute the workflow. The input row is stored so it can be read by the processors connected to the workflow input. Then the method delegates calculation of the result to the output processor `p` by calling `p.getResultValue()`. A processor fetches its required inputs from the workflow input or from its input processor with the `getResultValue()` call, after which it constructs a `ValueRow` with which it can `execute()` its associated RGL-Function. This way, values are pulled from the end of the workflow. The workflow processors recursively request their values from their respective inputs, calculating them where needed.

As a processor output may be used as input for multiple other processors, every processor remembers the result that was calculated with the `RGLFunction.execute()` call. This way, it needs to be calculated only once. If a processor or workflow is used to iterate, the stored result is removed so that a call to `getResultValue()` will request the fresh inputs and calculate the new result.

5.3 Implementation optimizations

This section discusses a number of optimizations that are implemented in RGL.

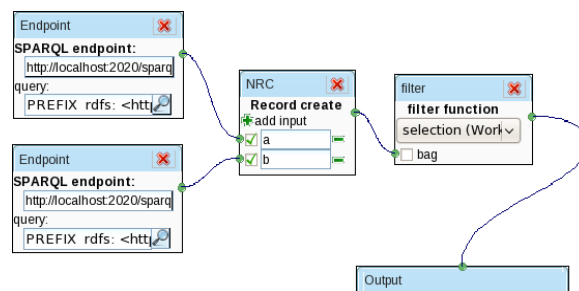
5.3.1 Pipelined bag iteration

The elements of a bag are generated by the bag iterator. It gives values one by one, until all elements of the bag have been produced. Figure 5.3 shows some of the bag implementations that exist.

The functional nature of the RDF Gears language makes that algorithms are implemented by concatenating functions in a pipeline. Take for example the workflow from figure 3.2c, which we can write in NRA as $map(g) \circ map(f) \circ h$. A naive implementation would calculate h , then the bag that results from $map(f) \circ h$, and then

To overcome these problems, RDF Gears applies an aggressive pipelining mechanism in which the bag contents are generated exactly when needed. By implementing the class `StreamingBagValue`, `Bag` implementations only need to implement the element generation mechanism. They do not need to be concerned with storing these results for later retrieval. As an example we explain the working of the `FilteringBagValue`, implemented by the `Filter` function implementation. It takes an input bag and outputs a bag F_{out} that contains all elements for which a certain qualification function returns `True` (see section 3.6.3).

Another example is the `BagUnion` implementation defined by the function that implements the NNRC bag union operator. A `BagUnion` instance references two bags, and is defined by the bag-union of the two. The iterator it provides simply iterates over all elements of the first bag, and then over all elements of the second bag. These are given to the user in a fire-and-forget way and not stored by the `BagUnion` itself. If the `BagUnion` result is iterated twice, the elements obtained by iterating over the input bags will be cached. This is done using a generic mechanism explained in section 5.3.2.



A final example is the `MappingBag`, which implements the RGL iteration mechanism when one or more processor inputs are marked for iteration. Consider figure 5.6.

Inputs a and b of the record creating processor are marked for iteration and thus creates a Cartesian bag-product from the bags A and B delivered by the SPARQL queries. The `MappingBag` iterates over the input bag with a nested loop, so if bag A is in the outer loop then A iterated once and bag B is iterated $|A|$ times. The resulting bag will contain $|A| \cdot |B|$ elements, causing the bag size to explode – conceptually, at least.

Thanks to our pipelined implementation, the non-materialized bags use constant $O(1)$ memory even if a consequent consuming operation (a filter, in the example) requires the bag with size $|A| \cdot |B|$ as input. The filter function just requests elements from the `MappingBag` iterator one by one. When an element is requested, the `MappingBag` generates the next combination of inputs for a and b from the input bags and create a record for the output bag. This record is then processed by the filter.

The execution of a pipeline of 10 bag mapping operations does not require storing any of the bags in memory. Instead, the implementation just executes the 10 element transformation functions on the first element when it is requested from the result bag, after which it can be written to the program output. Then the 2nd element is processed, etcetera. This makes it possible to work with very large bags that do not fit in memory.

5.3.2 Bag materialization optimizer

The implementations of `BagValue` provide a fire-and-forget iterator over their elements. They do not need to store the values, as bags are often used in a pipelined manner. When a bag is iterated more than once, however, it is advantageous to remember the produced bag elements in order to not generate them again. A typical example would be a workflow where the output of one processor is used as input for two different other processors. Another example is the workflow of figure 5.6, where the record creation operator iterates over the output of the SPARQL endpoint. If the top query yields $|A|$ elements and the results of the query results are not cached, the endpoint of the bottom query is queried $|A|$ times instead of only once. As executing a remote query is very costly, this is unacceptable. So fire-and-forget behavior is desirable to reduce the memory consumption, but in some cases the materializing behavior is desired to reduce the runtime.

Therefore RDF Gears implements an optimizer to determine what bags must materialize and what bags must fire-and-forget. It determines whether a bag will be iterated once, or possibly multiple times. Bags that are proven to be iterated only once are treated with the pipelined fire-and-forget mechanism described in the previous section. Bags that are possibly iterated more than once will have their elements stored internally in order to make consequent iterations cheap. Generating the elements still happens in the pipelined way described above, thus potentially saving the time of computing the unnecessary elements (i.e. when the bag is not iterated entirely).

The class `engine.Optimizer` optimizes the workflow by determining what processors produce a bag that may benefit from this materialization. In a minute we will describe the optimization rules. We will first explain how the `MaterializingBag` (figure 5.3) stores the elements generated by a bag.

If a processor is marked to materialize its bag output, it wraps the regular `BagValue` implementation it returns with a `MaterializingBag`. The `MaterializingBag` creates an iterator for the wrapped bag and does not maintain a pointer to the bag itself. This guarantees that the bag will only be iterated once. Iterators instantiated for the

`MaterializingBag` will gracefully share the iterator over the original bag and share an internal list of materialized results. When one of the iterators must produce a result it generates a result from the shared internal iterator only if a new result for that iterator is not already available in the materialized list. The list can be memory or disk-based, depending on the implementation of `ValueFactoryImpl#createBagBackingList` described above. The results are still generated in the pipeline, on request – but once they are computed, they are remembered.

This implementation neatly separates the concern of generating the values in a bag from the concern of caching them. The optimizer enables bag materialization only for those bags that may actually benefit from it, thus maximally exploiting the pipelined implementation, as follows.

Optimization rules

Assume a workflow hierarchy in which a well-typed workflow $W = (I, o, P, R, M, F, \sigma)$ is reused (W may also be the 'root' workflow). For every processor $p \in P$ the type-checking rules can determine its outputs type. Assume it is of type **Bag**(T) for some type T . Then the optimizer determines a truth value for s_p , (s for *streaming*) which is true iff the bag will be iterated at most once. If s_p , then a `MaterializingBag` is not needed; otherwise it is used, as the bag *may* be iterated multiple times, and materializing it will save the engine from calculating the bag contents twice.

We define s_p for any $p \in P$ that outputs a type **Bag**(T) as follows:

$$s_p = \begin{cases} x \vee \neg w & \text{if } (p, o) \in F \\ r \wedge ((i \wedge \neg m) \vee (\neg i \wedge f)) & \text{otherwise} \end{cases} \quad (5.1)$$

If $(p, o) \in F$ then p is linked to the workflow output port; it is the output processor. The definition of x and w is:

- $x \leftrightarrow I = \emptyset$. That is, the workflow does not have any inputs and can thus not be used in an iteration. This is the case when a workflow is executed by the engine, as the engine cannot execute workflows with unbound input ports (binding the input ports requires nesting it in a larger workflow W' with $I' = \emptyset$). Although the workflow may be reused by a processor p_w of another workflow, and p_w may in turn be read by multiple other processors, the materialization strategy for p_w will be determined separately. So although at one level a processor result may be streamed, it will be materialized on a higher level if necessary.
- If $\neg x$ then W must be a nested workflow, as otherwise the engine cannot evaluate the result. Assume it is reused by $W' = (I', o', P', R', M', F', \sigma')$ with some $p_w \in P'$ for which $\sigma'(p_w) = W$. Then we define $w \leftrightarrow \exists k : (k, p_w) \in M$, that is, the processor nesting W has one or more input ports that are marked for iteration. If w , then every bag output by p will be collected by the iteration mechanism of p_w to ultimately produce a value of type **Bag**(**Bag**(T)). The inner bags can be read more than once by any downstream processor reading from p_w . Therefore, the output of W is only guaranteed to be iterated once if w is false.

For any processor $p \in P$ with $(p, o) \notin F$ (i.e. p is not the workflow output processor) we define $B = \{(k, p_r) \mid \exists p_r \in P, k : (p, (k, p_r)) \in F\}$ (i.e. the set of processor input ports that read the output of p). Then we define r as:

- $r \leftrightarrow |B| = 1$. That is, there is exactly one processor input port that *reads* the output of p . If $\neg r$, multiple processors read the output of p_A and it may thus be read (iterated) multiple times.

We will define i (for *iterate*), m (for *more*) and f (for *function*) for those cases where r is true (as in the other cases their values do not affect the evaluation of the expression). We assume $B = \{(k, p_r)\}$, allowing us to refer to p_r in the following definitions of i , m and f :

- $i \leftrightarrow (k, p_r) \in M$. That is, the processor port reading from p is marked for iteration.
- $m \leftrightarrow \exists k_m : k_m \neq k \wedge (k_m, p_r) \in M$. That is, there is at least one more port in the reading processor that is marked for iteration, other than the port reading from p .
- f is true if the processor function of p_r is known to iterate over the input-bag maximally once, and input value is not copied to the output of p_r and not nested within that output. Examples of functions that meet these requirements are the filter function, the bag union operation, the bag flatten operation, the group operator, the categorize operator, a function that will determine the bag size, or determines whether a given element is contained in a bag.

A possible improvement (currently not implemented in our optimizer) can be achieved for the scenario $r \wedge (i \wedge m)$ where a processor is iterating over multiple inputs, such as in figure 5.11. In this scenario one of the iterated inputs of the middle processor is in the outer loop and need not be materialized as it is iterated only once. Note the similarity to the classical cost-based optimization technique for nested loop join, which iterates the largest bag in the outer loop.

Furthermore, our current optimizer may produce false positives in the materialization mechanism when workflows use complex intermediate datastructures that are deeply nested (i.e. bags of bags, or records with bags). Our current ruleset already proves quite effective, as we will show in sections 5.6.1 and 6.5. However, it may be improved by not assuming that a bag must be materialized if the reading processor *can* iterate over it multiple times, but by verifying this with a more sophisticated algorithm that verifies this for the whole workflow, similar to the typechecking mechanism. Note that this improved optimizer must then take into account the function semantics, which the current optimizer does only in a limited way.

5.3.3 Lazy evaluation of expressions

An aggressive lazy evaluation mechanism is implemented that delays evaluation of values as long as possible. Next to the fact that the pipelining implementation calculates bag elements while they are being iterated, the values within a bag or record are not evaluated even if they are used to construct a complex data structure. So although

conceptually the workflow passes around values between the workflow processors, the implementation passes around (function, inputrow) tuples. These represent a value, but are only evaluated when they really need to be, e.g. because it is serialized, used to determine control flow, bag filtering decisions, etc.

When a processors' output is requested, it requests the results of the processors connected to its inputs $\{k_1, \dots, k_n\}$ constructing input row $r : \{k_1, \dots, k_n\} \rightarrow \mathbb{C}$ (see section 5.2.3). Instead of executing the processors RGL function $f = \llbracket \sigma_p \rrbracket$ over the input row and returning the evaluated result of $f(r)$, the processor instead just creates a `LazyRGLValue` instance that contains

- the inputs given by the source processors: $r : \{k_1, \dots, k_n\} \rightarrow \mathbb{C}$.
- the function defined by the processor that produced the value: $f : \mathbb{C}^{\{k_1, \dots, k_n\}} \rightarrow \mathbb{C}$.
- a cached evaluation result, if the value has already been evaluated.

A simple example of where this is useful is that of figure 3.2b. The input to record field B does not need to be evaluated, as the projection of field A will discard that result anyway. The record creation processor reads its two inputs f and g (which are lazy values, too). It uses these to construct a tuple $[A : f, B : g]$, which is then input to a project function that yields a lazy value representing the expression $\pi_A [A : f, B : g]$. This expression is evaluated only when needed, e.g. if the expressions is part of the workflow output (the result must be printed to screen), when it is needed for the flow control (a boolean value is needed in an if-statement) or when it is otherwise required to determine the workflow output (e.g. it is a sub-expression of the workflow output, or it represents a boolean value used as a bag-filter qualification).

The implementation works as follows. When the implementation of the π_A operator is executed (note that this does not happen until that result, in turn, is required!) it fetches the value named `record` from the input row (remember we work with named inputs). Then the function simply returns the result of `r.asRecord().get("A")`. If `r` is a `LazyRGLValue` instance this will internally evaluate the function over its inputs, cast it to a `RecordValue`, and return the result. If the value is already `RecordValue` instance, the `asRecord()` is simply a syntactic sugar for class casting. The `get("A")` projects the field. Note that the result of g (under record field B) was an instance of `LazyRGLValue` that was never evaluated and is now lost.

This way, the lazy evaluation mechanism is not visible to the developers of RGL functions. They can simply treat the values as if they are real, using the casts `asRecord()`, `asBag()`, `asLiteral()`, `asGraph()`, etc.

Note that we have shown previously that static query rewriting rules from NRC can be applied to achieve optimizations for the example discussed. Although implementing these rules is not within the scope of this thesis, it would be desirable to incorporate these in RDF Gears. With such query rewriting the lazy evaluation would no longer give a benefit for the example given here. However, as long as that optimizer is not there, lazy evaluation saves a lot of unnecessary evaluation. Furthermore, it is a runtime optimization that can achieve performance improvements that are not easily achievable with static optimization alone. For example, take a conditional control flow that must be evaluated for every element of a bag; query rewriting is then not possible as the 'wiring' may differ per iteration. Another example is when operators are not

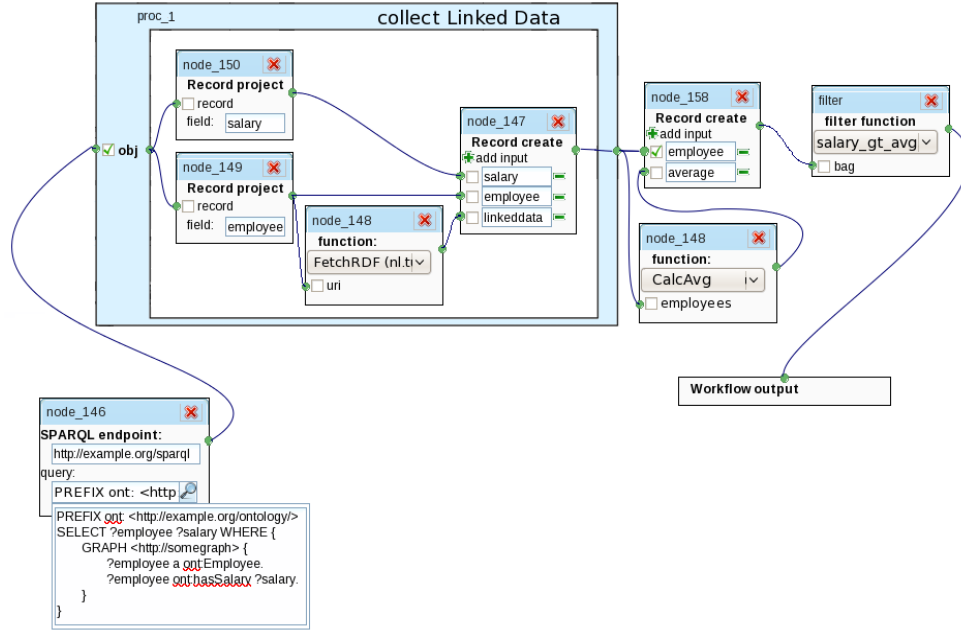


Figure 5.7: A workflow that benefits from lazy evaluation

transparent to the query optimizer, as shown in figure 5.7. It is loosely inspired by Buneman et al. [8] and modified for RGL.

The example shows how a bag **Bag(Record($\langle employee : \mathcal{U}, salary : \mathcal{L} \rangle$))** is fetched from a SPARQL endpoint and a **Bag(Record($\langle employee : \mathcal{U}, salary : \mathcal{L}, linkeddata : \mathcal{G} \rangle$))** of values is constructed using the *employee* and *salary* data from a SPARQL query and by downloading more data on the employee by fetching Linked Data from the *employee* URI. Furthermore an average salary is computed for those employees with the *CalcAvg* function. Assume this function is implemented as a custom function and the query optimizer cannot analyze its semantics. If the *avg* value is calculated, all employee records are now combined with this value in a structure **Bag(Record($\langle average : avg, employee : \text{Record}(\langle employee : \mathcal{U}, salary : \mathcal{L}, linkeddata : \mathcal{G} \rangle)$))**). Then the bag is filtered with the constraint $x.employee.salary > x.average$ for each element x .

Static query optimization cannot push the filter further back, behind the nested *proc_1* processor. Although in this case the *CalcAvg* function only needs the *salary* field of the input values, the *CalcAvg* function is not known to the optimizer and for all it knows, the function may require the *linkeddata* and *employee* fields.

The lazy evaluation technique now helps out. Fetching the employee Linked Data requires an expensive HTTP query. As those employees that do not qualify for the $salary > avg_salary$ constraint will be filtered out the bag, the HTTP-request is wasteful for those employees. The lazy approach would represent the graph as a tuple $(FetchRDF, \langle uri : \mathcal{U} \rangle, \cdot)$ (the third value is the cachedResult which is initially not available) and would not perform the required HTTP-request before the value is cast to

a graph with the *asGraph()* method, by some operator iterating over the filtered bag. Thus, no HTTP query would ever be fired if the employee is removed by the filter.

Thus, any static query optimization techniques that will be implemented in the future would ideally be combined with the lazy evaluation techniques that are currently implemented.

5.3.4 Field Index Mapping

RDF Gears works with the Named Nested Relational Calculus and with named function inputs. Rows over values (mappings from fieldnames or function input names to RGL values) are thus a prominent concept in the language. This is very flexible and intuitive for users as the fieldnames and named function inputs serve a self-documenting purpose.

However, administering such mappings for millions of values and function-calls can place high demands on system resources. For example, the Java Map implementations such as *HashMap* and *TreeMap* need to do significant administration when they are filled (rehashing, tree balancing) and are relatively memory intensive. The *HashMap* implementation for example administers parameters like thresholds, load-factors, *modCount* and *size* and it references a sparsely populated array of *Entry* objects, each consuming memory. This overhead is a significant waste of memory. Although the Java Map implementations are very fast, their memory consumption is not optimized for millions of very small Maps that all use the same keys.

Instead of maintaining a *Java Map<String, RGLValue>* for every record and function call, RDF Gears uses an approach that we call Field Index Mapping.

The Field Index Mapping approach works as follows. Due to the static typing of RGL, the number of record types and function signatures is limited by query size. RDF Gears therefore deals with a limited set of different types, and for each type it creates a *FieldIndexMap* that internally maps a fieldname to an array index. Records and named-argument function inputs are implemented in a *FieldMappedValueRow*. Instances of this class internally store all values in an array instead of a Java map and instances of the same RGL-type can reference the same *FieldIndexMap*. When a *Record* needs to project a field *A*, it can consult its *FieldIndexMap* to find out in what index of its own array the value for *A* is stored. RDF Gears features a *HashMap* based implementation of *FieldIndexMap* and an implementation based on array scanning. Section 5.6.2 will show that these provide a 60% reduction in run time and peak memory usage.

5.3.5 Buffered, streaming serialization

The (possibly complex) output value of an RDF workflow execution must somehow be serialized to display it or dump it to a file. Two serialization mechanisms are currently supported: one for serializing to an XML format, and one for a more human-readable but not parsable format called 'informal'. The third format is 'none', which will not serialize the value but force the necessary computational work required to calculate the value entirely, as would be needed for serialization. It does so by recursively evaluating the contents of the complex output value and is used for purposes of performance testing.

The serialization and recursive evaluator mechanisms are implemented with the Visitor pattern in the package `nl.tudelft.rdfgears.rgl.datamodel.value.visitors`. This pattern separates the concerns of serialization from the implementation details of the RGL values. This means that value implementations by third parties (such as the various bag implementations that RGL functions deliver, to optimize performance for that particular function) need not be concerned with such functionality. Also, implementing new serialization mechanisms (such as binary serialization for efficient storage of RGL values on disk) does not require any modifications to the implementation of RGL values.

The pipelined implementation of bags makes RDF Gears capable of processing values that would not fit into main memory. RDF Gears extends this pipelining philosophy by implementing the serialization classes in a pipelined way. This means the serialized value is also never kept in memory, making it possible to create serializations whose size exceed that of the main memory. Of course, output is buffered to reduce the overhead of I/O system calls.

5.4 The implementation of RGL core functions

This section discusses the implementation of a few RGL functions that were defined in section 3.6.3. These illustrate the particular implementation approach for that function, but also illustrate how the functions fit in the engine architecture.

5.4.1 Bag Group function

The BagGroup RGL function provides a way of transforming results of a SPARQL SELECT query into an NNRC structure that better resembles the format of the original RDF data without the need to repetitively fire SPARQL queries on an RDF datastore.

It is implemented in the class `nl.tudelft.rdfgears.rgl.function.core.BagGroup`. It is defined for a set of grouping fields $\{k_1, \dots, k_m\}$ and accepts any bag of type **Bag**(**Record**($\langle k_1 : \mathcal{V}, \dots, k_m : \mathcal{V}, \dots, k_n : \mathcal{V} \rangle$)).

It iterates over this bag only once, creating an in-memory list of the records. This list is then sorted¹. When iterated, the nested results are generated on-the-fly from the sorted list. Note that `GroupedBag` defined in the function directly extends `BagValue`. The `StreamingBagValue` functionality (providing an optional materialization wrapper) is not needed, as the class itself guarantees that it only iterates the input once and the records in the input bag are internally materialized.

The cost of sorting is assumed to be small compared to query execution and, in particular for remote queries, downloading of the query result. Note that the Bag Group method is intrinsically based on the bag-of-records type that features redundant data to simulate the nested collections that we distill from them. For a different approach, which fires multiple SPARQL queries instead of one in order to reduce the transfer of redundant data, see [2].

¹As of JVM7 Java uses Timsort having worst-case complexity $O(n \cdot \log(n))$ and best-case $O(n)$ [25]. This is attractive as results may be delivered in-order by the SPARQL engine.

5.4.2 Bag Categorize function

The Bag Categorize RGL function classifies elements in a bag. It creates a record with a field for every category, containing a bag with all elements in that category. It is implemented in the `BagCategorize` class that returns a record with instances of the class `CategoryBag`. An example that is inspired by Silk is shown in figure 5.8.

The bag categorize function receives a bag of records of the form $[lmbd : L, dbpedia : D]^2$. Assume the *categorizeLink* function defined for the categorizer is a workflow that determines the similarity of two elements, e.g. by querying other data sources, or by using some similarity metric like string-similarity. It returns for every record a Literal *accept* if the two elements are considered identical, *verify* if the elements are similar enough to justify manual verification, and *reject* if the elements are considered unrelated. The categorizer will then output a record of the form $[accept : B_{accept}, verify : B_{verify}]$ with B_{accept} and B_{verify} having the same type as the input bag (the elements for which the categorizer returned *reject* are lost, as the workflow image shows this category is not configured).

The implementation of the Bag Categorize function does use pipelining, but builds an internal list for every category that may grow to the size of the input bag. To illustrate this, assume that the *accept* field of the record output by the example is iterated first (e.g. for the purpose of serializing it to disk or screen). During the iteration over B_{accept} the Bag Categorizer generates the contents in a pipelined fashion by iterating over its input bag, returning the elements for which the *categorizeLink* function returns *accept*. However, it will encounter many elements for the *verify* or *reject* category. The elements that qualify as *verify* are kept in memory, to be delivered when B_{verify} is iterated. This prevents that for every generated group all elements need to be classified again.

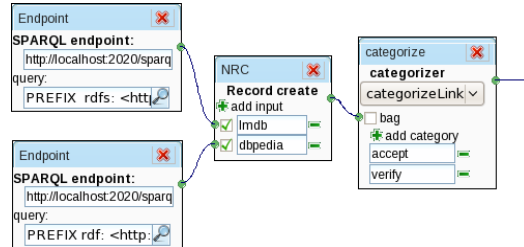


Figure 5.8: Example of using the bag categorize function, inspired by Silk.

5.4.3 SPARQL query function

The implementation of the RGL SPARQL query function deserves some special attention. Figure 5.9 shows the class `rgl.function.sparql.SPARQLFunction`. It implements the delegator pattern. The `SPARQLFunction` picks the delegatee (implementation of `SparqBehavior`) based on the availability of an endpoint URI and the nature of the given query (SELECT or CONSTRUCT).

²More specifically, we assume the complete type of the bag is **Bag**(**Record**(*lmbd* : (**Record**(*movie* : $\mathcal{V}$, *label* : $\mathcal{V}$), *dbpedia* : (**Record**(*movie* : $\mathcal{V}$, *label* : $\mathcal{V}$))))).

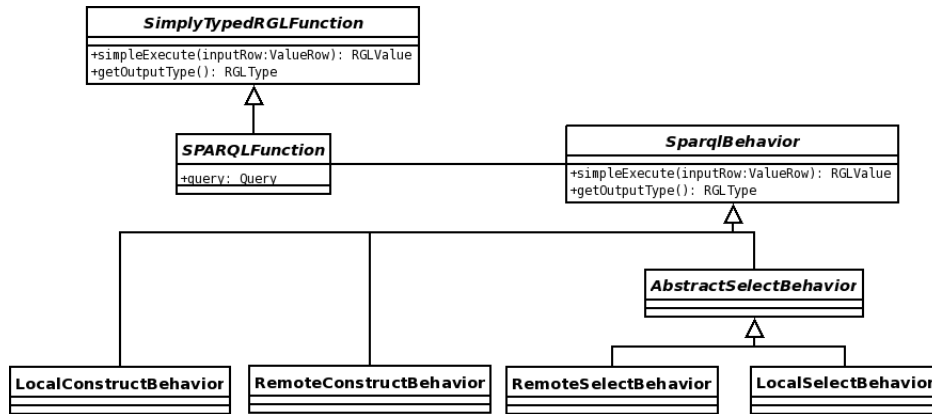


Figure 5.9: Class diagram for the SPARQLFunction implementation

Query paging

Some popular SPARQL implementations (e.g. Virtuoso) will never return more than a certain number of elements (e.g. 2500). This makes it impossible to fetch more than 2500 results with a single query, even if there are more than 2500 matches for the query. To fetch all results of a SPARQL query from such a remote endpoint, RDF Gears implements query paging that is also implemented by Silk. Take for example the query of figure 5.10.

```

SELECT DISTINCT (str(?label_lang) AS ?label) ?mov
WHERE {
  ?mov rdf:type dbpedia:Film.
  OPTIONAL {
    ?mov dbpedia:director ?dir.
    ?dir rdfs:label ?label_lang.
  }
}

```

Figure 5.10: A SPARQL query that may yield more than 2500 results

The user can specify the maximum number of results a SPARQL endpoint can produce in one query (e.g. 2500). The `RemoteSelectBehavior` class repetitively queries an endpoint with modified LIMIT/OFFSET, until less than 2500 results are delivered. If the total number of results in the SPARQL endpoint is 7000, these will all be fetched in three requests. The first request fires the query with `LIMIT 2500 OFFSET 0` appended. The second request appends `LIMIT 2500 OFFSET 2500` to the original query. The third request does `LIMIT 2500 OFFSET 5000`. Then the SPARQL endpoint will return only 2000 results, indicating that there are no more results on the endpoint.

Although effective, it must be noted that this behavior is not defined by the SPARQL standard³. However, in practice the endpoints use some implicit ordering

³<http://www.w3.org/TR/rdf-sparql-query/#modOffset> states "Using LIMIT and OFFSET to select different subsets of the query solutions will not be useful unless the order is made predictable by using ORDER BY.". But ORDER BY is not used as this is slow in the Virtuoso implementation.

that makes this work for both RDF Gears and Silk. For CONSTRUCT queries the query paging should never be applied as the semantics of OFFSET in CONSTRUCT queries is not defined. Experiments have shown that the Virtuoso and ARQ SPARQL implementations give very different and unpredictable behavior.

The modification of the queries is not done by string replacements, but by parsing the query and modifying the SPARQL syntax tree to contain a LIMIT and OFFSET (see the `com.hp.hpl.jena.sparql.serializer.QueryLimitRewriter` class implemented for RDF Gears). This guarantees that the rewriting is correct even if strings in the SPARQL query contain these keywords.

Finally note that the option `greedy_loading_of_remote_queries` of the RDF Gears config file can be useful when large datasets are retrieved. If the value is true, RDF Gears loads paged data from SPARQL endpoints in a non-pipelined manner. This brings failures in loading the data (e.g. when the SPARQL endpoint goes down) to light as soon as possible, instead of halfway the time-consuming execution of a complex workflow.

Prebinding variables

It is possible to specify a SPARQL query with free variables, and bind these variables with the input row of a processor. This is defined in section 3.6.4. For local SPARQL queries, RDF Gears implements this by using the `QueryExecution#setInitialBinding()` function of ARQ. However, ARQ does not implement this for remote query executions. It is therefore not currently functional in the implementation, but it is low-hanging fruit: prebinding of remote SPARQL queries can be enabled by providing a query rewriting mechanism in `QueryUtil.createRemoteQueryExecution()`, similar to the approach that was used for the OFFSET rewriting discussed above.

5.5 Extending RGL with custom functions

The architecture of RDF Gears makes it easy for third party programmers to provide their own RGL function implementations in Java. To do this, the programmer must specify a data transformation algorithm and an algorithm that manipulates the types.

The simplest way to implement an RGL function in Java is by extending the `rgl.function.SimpleTypedRGLFunction` class. It is used for functions that always receive and return the same type and that cannot deal with *null* values. The implementation of the transformation must be implemented in the `simpleExecute()` method, that translates a `ValueRow` to a new RGL Value. To illustrate the specification of the typing function (see definition 3.4.2) we take the `JaroSimilarity` implementation as an example. Its implementation of the `getOutputType()` method is just `return RDFType.getInstance()`, thus returning an object that represents the `RDFType`. So the range of the typing function contains only that element. The domain of the function, specifying for which inputs the function can be used in a well-typed workflow, is set with two calls in the constructor: `requireInputType("s1", RDFType.getInstance())` and `requireInputType("s2", RDFType.getInstance())` define that the `JaroSimilarity` function returns an `RDFValue` if it receives an `RDFValue` for

both the `s1` and `s2` inputs. If the implementation of the `simpleExecute` method is called, the inputs are guaranteed to be non-null RGL values (although bags and records may *contain* null values). If a workflow passes a *null* value to a processor with a `SimplyTypedRGLFunction`, the function is not called and the processor output is set to *null*.

This way, the `SimplyTypedRGLFunction` simplifies function implementation where possible. To implement polymorphic functions, the `AtomicRGLFunction` is extended directly. This provides flexibility when desired. For such functions, it is worth to study the implementation of the RGL core operators.

The RGL data transformation of an RGL function can be implemented using the RGL API. It provides a conceptual view over the RGL values and does not in any way require understanding of the engine optimizations. For functions that return a bag value, the user merely needs to provide a mechanism to iterate over this bag once. So the implementation does not actually *construct* the bag, but does specify an iterator that defines it. The bag implementations are thus agnostic to the way they are materialized by RDF Gears for multiple iterations. This makes implementation of third-party functions simple and allows them to seamlessly integrate with the typechecking mechanism and to benefit from the engine optimizations.

5.6 Evaluation of the implementation optimizations

This section will evaluate the performance gained with the pipelining behavior in combination with the optimizer. This not only shows that the pipelining indeed works correctly in the implementation, but it also illustrates how the optimizer is necessary to determine when pipelining is *not* beneficial – that is, when intermediate results should be materialized in memory.

Then the benefit of the Field Index Mapping technique is evaluated, showing that this not only benefits the memory usage, but also shortens the run time up to 60%.

5.6.1 Evaluation of pipelining and the optimizer

The time and space complexity of the bag implementation was verified with three configurations. The first configurations materialized all bags as if pipelining was not implemented. The second configuration used the pipelining approach with fire-and-forget iteration for all bags. The third configuration used the optimizer to determine what processor should deliver materializing bags and what processor should deliver fire-and-forget bags. It shows that although not all bags are suitable for streaming, the optimizer is effective in deciding which processors can and which processors cannot be streamed.

For this experiment we used the workflow of figure 5.11. The results are summarized in figure 5.12. For complete data, see appendix C.1. The data clearly show that for this trivial workflow the RDF Gears optimizer gives a huge performance win by creating bags that use $O(1)$ memory where possible, and by materializing bags (using $O(n)$ memory) where necessary to keep the processing time within acceptable bounds.

The processors `queryA` and `queryB` in the workflow execute SPARQL SELECT queries yielding bag *A* (of type **Bag(Record(r_a))** for some row over RDF values r_a) and bag *B* (of type **Bag(Record(r_a))** for some row over RDF values

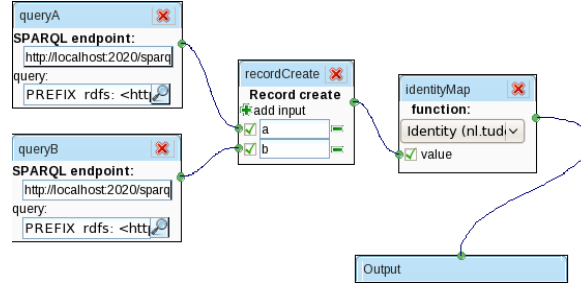


Figure 5.11: The `testStream` workflow used for the pipelining experiment

r_b). The `recordCreate` processor iterates over both inputs creating a bag with type **Bag**(**Record**($\langle a : r_a, b : r_b \rangle$)) containing $|A| \cdot |B|$ elements. In our experiment we let $|B|$ vary from 0 to 8000 elements by setting a LIMIT in the query. To really measure the bag complexity and not that of the algorithm, $|A| = 2500$ is fixed and serves to increase the bag size by a constant factor.

Thus the algorithm has $O(|A| \cdot |B|)$ time complexity and in our experiments the bag size is linearly affected by $|B|$ (figure 5.11 shows $10 \cdot 10^6$ elements for $|B| = 4000$ and $20 \cdot 10^6$ elements when $|B| = 8000$).

Materializing all bags

In the materializing setup all bag results are stored in memory as soon as they are generated. Our test setup did apply pipelining (figure C.1a shows the gradually rising memory usage). However, by forcing all bags to materialize their results, the peak memory usage is equivalent to a non-pipelining architecture where all bags would be entirely generated before passing them to the next operator.

All bags now consume $O(n)$ space. The total space required for workflow execution is $O(|A| \cdot |B|)$, which requires 6 GB of main memory to materialize $20 \cdot 10^6$ elements. Time complexity of iterating a bag is $O(|A| \cdot |B|)$ ⁴.

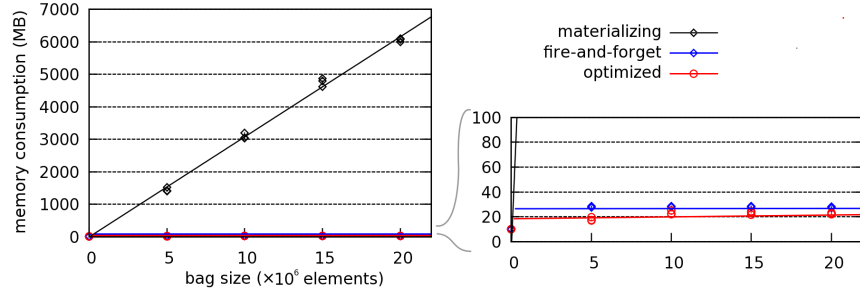
Fire-and-forget in all bags

In the second setup no bags were materialized at all, but instead results were generated in the pipeline with a fire-and-forget approach.

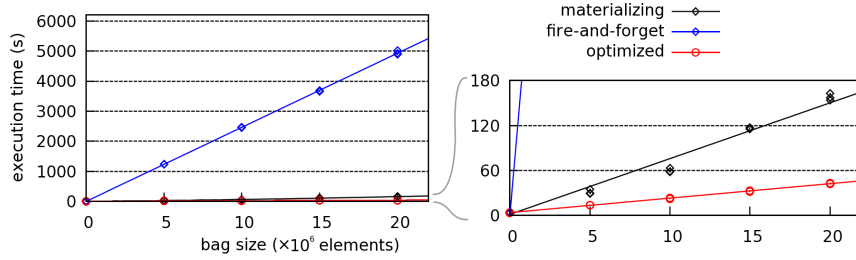
With this approach the total space required for workflow execution is theoretically $O(1)$, assuming all bags can work in a streaming way (i.e. database results are download in a streaming fashion, etc). This is at least true for the bag of size $|A| \cdot |B|$, as the blue line in figure 5.12b shows constant memory usage⁵.

⁴Actually the materializing result seemed to show a mildly polynomial behavior in some experiments, but there is insufficient data to support or explain this.

⁵The measurement uncertainty is caused by the JVM garbage collector, as shown in figure C.1. This is due to the fact that we gave the JVM 32 MB of space (with the flag `-Xmx32m`). Experiments have shown that with $|B| = 8000$, the workflow can actually be run within 16 MB of memory with both the streaming and optimized setup. For the materializing setup, on the other hand, even 5000 MB is insufficient to evaluate the workflow.



(a) Space complexity of the bag is $O(n)$ when a bag is materialized. With the fire-and-forget and optimized strategies it is $O(1)$. The measurements apply to the bags with $n = |A| \cdot |B|$ as the smaller bags do not significantly contribute to memory consumption.



(b) Time complexity of workflow execution is $O(|A| \cdot |B|)$ which is linear with constant $|A|$. The optimizer materializes only where necessary and thus prevents repetitive generation of a bags A and B . This reduces run time for this workflow by 99% compared to fire-and-forget. By eliminating unnecessary materialization, it is 70% faster then the materializing approach.

Figure 5.12: The optimizer reduces both execution time and memory consumption.

The problem is that the runtime explodes because the bag contents are computed again for every iteration. In this trivial example, the Cartesian bag-product requires many iterations over one of the query outputs. This means that processor *queryB* executes $|A|$ queries, instead of a single one. In the best-case scenario the cost of executing *queryB* a linear $C_1 \cdot |B| + C_2$. The cost of iterating B is $C_3 \cdot |B|$. The total workflow execution then has time complexity $O(|A| \cdot (C_1 \cdot |B| + C_2) + |A| \cdot (C_3 \cdot |B|))$ which is still $O(|A| \cdot |B|)$. However, the constant C_1 (cost of downloading and parsing a query result) is so large compared to C_3 (cost of fetching a materialized element from memory) that this creates a huge constant slow-down⁶.

For other workflows this constant factor may be smaller, depending on the computational cost of generating the bags. However, it is also desirable to save duplicate work in scenarios with a smaller constant penalty of regenerating the result. Examples are the operation of fetching Linked Data from the Web (it is desirable to resolve all URI's only once instead of once for every iteration over the bag) or just the calculation of the Jaro similarity of two strings (used in section 6.3).

⁶In this particular case it may also upset your database administrator. Tell him to do query caching!

Optimizer decides per processor

The third setup combines the best of both worlds: the optimizer configures every processor explicitly to either create bags with fire-and-forget behavior, or to create bags with materializing (caching) behavior.

The optimizer decides to only materialize bags A and B . The bags of size $|A| \cdot |B|$ are streamed in a fire-and-forget way:

```
DEBUG [main] (Optimizer.java:77) - bags output by proc identityMap will
    be streamed
DEBUG [main] (Optimizer.java:77) - bags output by proc recordCreate will
    be streamed
DEBUG [main] (Optimizer.java:80) - bags output by proc queryB will be
    materialized
DEBUG [main] (Optimizer.java:80) - bags output by proc queryA will be
    materialized
```

The red line shows the optimized approach creates the lowest memory usage (figure 5.12a) and the lowest run time (figure 5.12b). It shows the bags output by the last 2 processors use $O(1)$ space. The workflow as a whole thus consumes $O(|A| + |B|)$ space instead of $O(|A| \cdot |B|)$ of the materialized approach. Due to the measurement uncertainty and small values for $|A|$ and $|B|$ this does not significantly contribute to the memory requirements.

Interestingly, the optimized setup is not only a 99% faster then the pure fire-and-forget approach, but also 70% faster than the materializing approach. This may have to do with the overhead of memory-management for such large amounts of memory⁷.

Note that although the workflow for our experiment uses an Identity operator, the workflow of figure 5.6 gives the same qualitative results, as does a workflow that omits the last operator.

5.6.2 Evaluation of Field Index Mapping

We evaluated the Field Index Mapping technique of RDF Gears by comparing three scenarios: one where no Field Index Mapping was used, one where Field Index Mapping was implemented by HashMaps, and one where Field Index Mapping was implemented with an array.

We used the workflow of figure 5.11. Because we aim at testing the memory consumption and computational cost of creating and indexing the value rows, we used the "materializing" setup described in section 5.6.1. Although we have shown in section 5.3.2 that this setup uses more memory than necessary, it also showed that materializing bags is sometimes still required. The memory occupied by values in a materialized bag cannot be garbage collected, and it is therefore desired to reduce their memory footprint.

The following setups were all executed three times, the average run time and memory usage is shown in figure 5.13. It is clear that Field Index Mapping significantly reduces the memory consumption as well as the execution time.

⁷The MaterializingBag internally uses one ArrayList that reallocates memory if it grows too big.

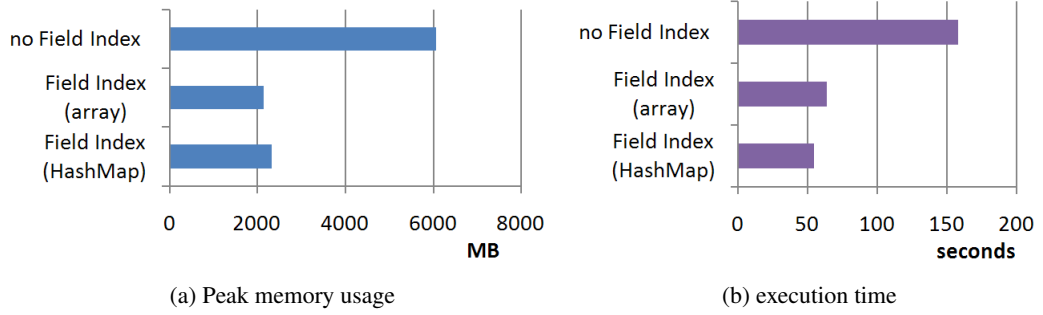


Figure 5.13: Field Map Indexing reduces execution time and memory usage

No Field Index Mapping

In the first setup, we ran the workflow without the FieldMapping technique: all RGL records and named function inputs contained their own `HashMap<String, RGLValue>`. We have seen in figure 5.12a that this requires over 6 GB of memory⁸ with $|A| = 2500$ and $B = |8000|$.

Field Index Mapping (HashMap based)

One implementation of the Field Index Mapping uses a `HashMap<String,Int>` to provide the mapping from fieldNames (String) to an array index (int). The significant improvement in memory usage can be attributed to the fact that a Java array consumes less space than a HashMap with the same number of elements. The improvement in execution time is explained with the fact that instantiating records and other value rows (function inputs) does not require instantiating and filling new HashMap instances. Instead, a new record just stores a pointer to an existing `FieldIndexMap` instance. Fetching a named element from a record still requires indexing the HashMap, as the record finds the right index in its internal RGLValue array by consulting the `FieldIndexMap`.

Field Index Mapping (array based)

Another implementation of the Field Index Mapping uses an array to store all the field names. It can map a field name to an array index by simply iterating the array until it finds the right field name. The index at which the field name is found is returned to the `RecordValue` instance, which uses the same index to fetch/store its internal value.

The memory usage was expected to be the same for these approaches, as in both setups the 20.000.000 records all reference only a single `FieldIndexMap` instance in both setups.

We suspected that the array-based Field Index Mapping would be significantly faster than the HashMap-based approach when indexing the small 2-element records of the example. The figures contradict this. The Java String class caches its computed hash code and the rest of the Java HashMap implementation is apparently very fast.

⁸The experiments in the previous section were all conducted without the FieldMap indexing technique.

It seems that HashMap is even slightly faster, but we do not consider the difference between the two Field Index Map implementations significant, given the variance of the data (see appendix C.2).

5.7 Conclusion

The RDF Gears Engine can execute the workflows defined in XML, as output by the user interface. It uses the Jena and ARQ libraries for RDF querying and manipulation. The engine supports the typechecking of workflows; when a workflow is ill-typed, a useful error message is produced that helps the user to find the problem location in the workflow stack. The command line flags of the engine executable can be used to specify the workflow to be used, the paths where it must be searched for, whether the workflow must be merely typechecked or actually executed, and what serialization mechanism to use for the result RGL value. The engine can either serialize the value in a machine-readable XML representation or with a more concise, human readable output that is useful for debugging workflows.

The engine is implemented with a flexible, extensible object oriented architecture. For example, the value serialization mechanism is entirely separated from the value implementations with the visitor pattern. Also, the bag implementations are separated from the materialization mechanism that they use. Thanks to this architecture it is possible to implement disk-based materialization mechanisms by adding a new Value-Factory and replacing the implementations of MaterializingBag. For efficiency, a new (binary) value serialization mechanism can be plugged in. The values implementations themselves need not be touched for any of these modifications.

The architecture of the engine is designed to be flexible and extensible. It applies a separation of concerns that allows new RGL serialization formats to be implemented without touching the RGL Value implementations. A function that returns a custom Bag value need not implement any materialization functionality, but is able to benefit from the global mechanism. This materialization mechanism can be changed, e.g. to implement a hybrid disk- and memory-based approach, without touching the optimized bag implementations of the operators.

Various optimizations are implemented that significantly reduce the runtime and memory consumption of workflow execution.

The aggressive pipelining approach reduces the space complexity of bags from $O(n)$ to $O(1)$. This reduction in space complexity makes it possible to process bags that would not fit in memory. A workflow that required 6 GB now consumes only 14 MB (figure 5.12a). Bags that are iterated multiple times are recognized by a static optimizer and will still receive in-memory materialization. In our experiments this optimization delivers a runtime improvement of 70% compared to a non-pipelining approach where everything is materialized, and up to 99% for an approach that is completely pipelining and does not materialize anything in memory (figure 5.12b).

A lazy evaluation mechanism is implemented that delays evaluation of values as long as possible. Next to the fact that the pipelining implementation calculates bag elements while they are being iterated, the values within a bag or record are not evaluated even if they are conceptually copied into another data structure.

The Field Map Indexing technique allows the RGL record implementation to store

its values in an internal array, by implementing the mapping of field names to array indices in a separate class. This way, records do not need store such a map but can instead share a Field Map Index. This reduces execution time by 60% and reduces memory usage by over 60% (figure 5.13).

Chapter 6

Case study: Comparison with Silk

This chapter evaluates RDF Gears by comparing it to Silk, using a typical Silk data integration algorithm. Section 6.1 introduces Silk and its implementation optimizations.

A typical data integration scenario implemented in Silk. Then section 6.3 shows how the same data integration can be implemented with RDF Gears. Section 6.4 discusses how RDF Gears is even more flexible and can be used to implement more complex transformations that cannot be expressed in Silk.

Section 6.5 evaluates the performance of RDF Gears by comparing it to Silk, after which the chapter is concluded.

6.1 Silk introduction

The Silk framework [29], which was briefly introduced in section 2.2, provides tools to generate links between data items based on user-provided link specifications. Silk is roughly based on the definition of instance matching given by equation 2.1. It uses fuzzy matching rules to calculate a score for every pair of elements of two datasets. This score is a probability value that a certain relation between the two elements exists. With configurable thresholds the links between entities can either be disqualified or materialized for verification or acceptance. For the (possibly) relevant pairs, new RDF triples are created to interlink the entities.

The Silk implementation is designed and optimized to perform well on large datasets. Silk is implemented on the Map Reduce framework to allow parallel computing. On a single machine it can run on multiple cores. By using multiple machines, computation can be distributed over a computational grid.

In its simplest version, Silk calculates the probability for all possible pairs between the two given datasets. Since recently, Silk features the *MultiBlock* method [28]. Instead of taking the Cartesian product over all elements of both dataset, this method assigns every entity of both datasets to one or more blocks before calculating the similarity values. It guarantees that entity pairs whose similarity exceeds the given threshold share at least one block. The Cartesian product between the entities from A and B is taken for every block, instead of for the whole datasets, after which the similarity is calculated for those pairs. This reduces the number of combinations over which the similarity calculation is performed. Thus it reduces the overall runtime significantly, without affecting the recall of the link generation.

6.2 A typical Silk transformation

The Silk 2.0 distribution comes with the example document `linkedmdb_directors.xml`. This use case creates `dbpedia:director` links between DBPedia movies and directors in LinkedMDB. This link specification was configured to query our local SPARQL endpoint and is shown in appendix C.3. We will explain it here, as we will implement the same algorithm in RDF Gears.

The Silk algorithm of the example queries an endpoint for all movies in DBPedia with the query `SELECT ?a WHERE {?a rdf:type dbpedia:Film. }`. It then fetches all directors in LMDb, the Linked Movie Database, with `SELECT ?b WHERE {?a rdf:type movie:director.}`.

The `<LinkCondition>` section specifies the fuzzy matching algorithm. For a pair of strings (e.g. names) the Jaro similarity metric [15] is used to compute a score between 0.0 (not similar) and 1.0 (identical). The example computes p_1 as the similarity of the director-label of the DBPedia movie with the directors' label in LMDb. It computes p_2 as the similarity of the DBPedia director label with the LMDb `movie:director_name` value. The probability that the director at hand is the director of the DBPedia movie is then defined $score = \max(p_1, p_2)$ by this LinkCondition. Note that multiple values $score$ can be computed for one (movie,director) pair as multiple `rdfs:label` and `movie:director_name` properties can be defined.

The `<Filter>` specifies that a link can be created for a movie `?a` if the single best value for $score$ exceeds 0.7. The `<LinkType>` defines that the link created is the triple `?a dbpedia:director ?b1`. If $0.7 \leq score \leq 0.9$ then the triple should be manually verified, if $score > 0.9$ it can be accepted (as per the `<Outputs>` section).

6.3 Equivalent transformation in RDF Gears

The Silk algorithm we just discussed is implemented in RGL by the workflow of figure 6.1. In red, the edge typing annotations are displayed. For conciseness, the notation $\{[k_1 : T_1, \dots, k_n : T_n]\}$ is used to denote the type **Bag(Record($\langle k_1 : T_1, \dots, k_n : T_n \rangle$))**.

At the start of the workflow, on the left, both datasets are queried. The DBPedia dataset is queried for the movies and label of its director, thus outputting a bag $\{[T_D]\}$ containing a record of type $T_D = [\text{mov}:\mathcal{L}, \text{label}:\mathcal{L}]$ for every director-label of every movie. The LMDb dataset is queried and returns a bag $\{[T_L]\}$ containing a record of type $T_L = [\text{dir}:\mathcal{L}, \text{dir_name}:\mathcal{L}, \text{label}:\mathcal{L}]$ for every combination of a directors name and label.

The *nestResults* processor applies the Bag Group function, grouping the director-labels by the movie with which they are associated. Over this result the workflow on the first nesting level iterates, executing its algorithm repetitively for every movie.

The workflow at the second level attaches a `score` value to each combination of T_D and T_L for this movie. This value is calculated by the innermost workflow which implements the Silk `<LinkCondition>`.

¹As the directors in DBPedia are reified (i.e. have their own URI), this particular example from the Silk 2.0 distribution could better be modified to create `owl:sameAs` links from DBPedia directors to LMDb directors. Note, however, that some datasets will not have reified entities on the source dataset (i.e. the director of a movie is merely defined with a string-literal). This would then justify the creation of a `?a dbpedia:director ?b` link.

The *selectBest* processor, executing the top-score function in the first nesting level, now receives a bag $\{ \{ [dbpedia:T_D, lmdb:T_L, score:L] \} \}$ containing the *score* values for this movie, one for every combination of its labels and all director-names and -labels of LMDB. It takes the element of the bag that has the highest score function, using the trivial record projection workflow of figure 6.2.

Finally, the best matches for all movies are categorized by *verifyAccept* processor at the end of the outer workflow, using the workflow of figure 6.3 to categorize the link as "verify", "accept" or "reject". The output format is similar to that of the ontology alignment format² used by Silk. It can be further processed with RDF Gears (e.g. to create `owl:sameAs` triples), or the RDF Gears XML serialization of the result value can be processed with external technologies such as XSLT.

6.4 Expressivity comparison

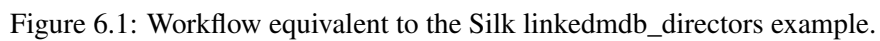
It turns out that RGL is capable of expressing the same data transformations that Silk can. Note that it was required to implement a *top-score* function for RGL as this is not (yet) defined as an RGL core function. This seems to be such a generic function that it can readily be reused for other workflows, and a generalized function that accepts the top *n* elements of a bag should possibly even be added as standard RGL function.

Although the RGL workflow and the Silk Link Specification describe the same data integration transformation, we must note that they do provide different results. This is due to the fact that neither of the algorithms have any defined behavior when a movie has multiple directors. For example, DBPedia specifies that the movie [http://dbpedia.org/resource/Black_Fist_\(film\)](http://dbpedia.org/resource/Black_Fist_(film)) has two directors, Timothy Galfas and Richard Kaye. Both have an equivalent entry in LMDB that has a 1.0 similarity score. The best match is arbitrarily selected, and Silk may accept another director than our workflow. Because both links have the same similarity score, there is no general difference in recall or precision for the two implementations.

In Silk it is possible to increase the limit size to find multiple directors for a movie, but then one DBPedia director label may be associated with multiple entries in LMDB. This is not what we want. In RGL, the workflow can be simplified as follows: remove the Bag Group operation (*nestResult* processor), the associated *project_{group}* operation (*project* processor in workflow at first nesting level) and the iteration-mark at input *D* of the workflow at the 2nd nesting level. The *top-score* function then finds the best LMDB match for every DBPedia director-label of a particular movie *m*. This will create one link between a movie *m* and its director *d* for every director-label specified for *m*. If different labels have the same match *d*, this duplicate is automatically filtered when the results are converted to RDF.

This brings to light the fundamental limitations of the expressivity of Silk. Although the fuzzy matching rules in the `<LinkCondition>` can be flexibly combined, the combinatorial algorithm that defines what items are compared can merely be configured with a few thresholds and limit parameters. Silk is fixed to the rigid combinatorial structure that combines entities in two datasets, and does therefore not provide

²<http://alignapi.gforge.inria.fr/format.html>



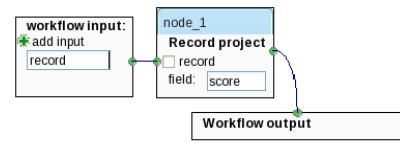


Figure 6.2: The trivial record projection workflow used by the top-score processor.

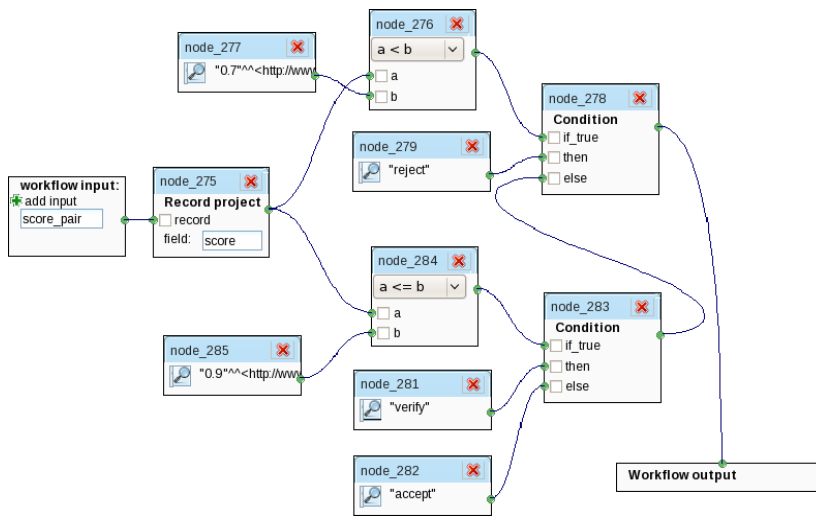


Figure 6.3: The workflow that categorizes a score as "accept", "verify" or "reject".

the expressivity for all interlinking scenarios³. This is particularly painful when Silk *almost* does what you want.

RGL, on the other hand, allows the user to flexibly specify the exact behavior of the integration process. RGL algorithms can be studied and modified without limitations to the complexity of the resulting workflows. The already expressive set of core operations can also be extended. Clearly, RGL is more expressive than Silk.

6.5 Performance comparison

The goal of our performance comparison with Silk is twofold.

The first goal is to assess the constant order overhead RDF Gears incurs to the example algorithm. Our generic interpreter passes values between the various operators, uses the generic NNRC datastructures, must determine itself where fire-and-forget iteration of bags is beneficial, etcetera. Silk on the other hand uses an iteration/combination algorithm that is known in advance and is merely configured with some parameters and a way to calculate the similarity between two given instances. RGL has a more

³For a more complex example consider also the scenario where more than 2 data sources are consulted to find interlinking relations, or the scenario where the similarity is determined by involving complex aggregates [45].

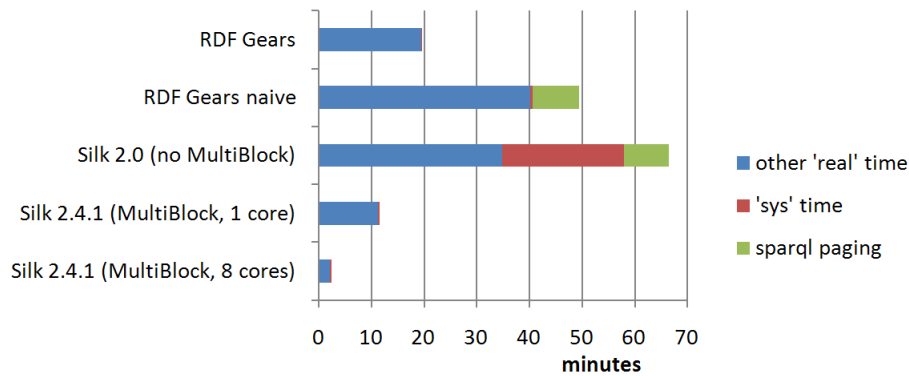


Figure 6.4: Execution time comparison between Silk and RDF Gears.

flexible, generic nature, which makes it harder to implement efficiently. A performance comparison shows the cost of this flexibility in the engine implementation.

The second goal is to see how much a data integration algorithm like that of Silk can be improved with a multi-core architecture and query optimization techniques like MultiBlock.

All experiments were performed on a single core, with the exception of the Silk 2.4.1 8 core experiment. The results are summarized in figure 6.4. The full length of the bars is the total ('real') execution time. Complete results are in appendix C.3.

6.5.1 RDF Gears

Two setups were tested. They are referred to as *RDF Gears* and *RDF Gears naive*. Both the RDF Gears configurations could be executed in 100 MB of memory.

RDF Gears

The first experiment ran the workflow of figure 6.1. It fetched 92.225 tuples representing (movie, director-label) records for 53.618 unique movies (a director can have different labels, e.g. in Chinese and English). Note that the `SELECT str(?label_lang) AS ?label` query part in the queryDBPedia processor eliminates labels that have an identical string-part (i.e. the same string-representation in different languages). The workflow is executed in 19 minutes on average.

RDF Gears naive

The second experiment did not do the elimination of duplicate labels in the query, nor in the workflow. DBPedia then returned 186.956 labels (about twice as much). As expected, the run time then grows proportionally with this input to 41 minutes. If SPARQL paging is enabled for RDF Gears, execution takes 8 minutes longer.

6.5.2 Silk Tests

Silk was tested both with version 2.0 and with the more recent version 2.4.1. There are two reasons for this. Firstly, the 2.4 version uses a newer LSL format and wasn't released when we designed the workflow and the tests. Secondly we wanted to compare RDF Gears with an implementation that does not use MultiBlock, and it cannot be disabled in Silk 2.4.

Silk 2.0 (no MultiBlock)

Silk used 250 MB where RDF Gears used 100 MB. We could not limit the JVM options of Silk, but the Silk implementation could probably be more sparing with more aggressive garbage collection.

Silk without MultiBlock was ran on a single core, completing execution in 66 minutes. RDF Gears produced the same links in 19 minutes and is thus 70% faster than Silk without MultiBlock.

It is remarkable that the Silk process was reported to use about 22 minutes of system time⁴, while RDF Gears used only 12 seconds. It is not clear why this is – it seems an unnecessary waste somehow. Furthermore, the Silk queries did not apply elimination of identical labels in the DBPedia query, giving it 186.956 combinations of a movie and a director label. This is a fair design decision as it is not strict SPARQL 1.0. There is no reason, however, to not eliminate such duplicates after retrieving the query results. It is not clear whether Silk 2.0 actually does this.

Now let us assume that we just forget about the 22 minutes of system time Silk seems to waste. Furthermore we give Silk 2.0 the benefit of the doubt by assuming it does not eliminate duplicate labels for a movie and compares the full 186.956 movie tuples (as noted above, this would be naive). Finally, we compensate Silk for the query paging it does although the endpoint does not require this (the time spent is largely idle time, waiting on the endpoint response). With these compensations Silk is given the royal benefit of the doubt. Silk is then 16% faster than the *RDF Gears naive* setup, as the blue bar in the image shows.

Silk 2.4.1 (MultiBlock, 1 core)

Silk 2.4.1 was executed with the MultiBlock method. This Silk version used up to 1500 MB of memory, unlike RDF Gears and Silk 2.0. It is possible that Silk can do with less if the JVM is configured accordingly, or that this amount is really required for the MultiBlock method.

On a single core Silk with MultiBlock finishes execution in about 11 minutes. This is almost twice as fast as RDF Gears on a single core, which we attribute to the MultiBlock method. So it can be concluded that the query optimization implemented by Silk is worthwhile for run time⁵.

⁴Reported as *sys* by the GNU *time* command

⁵Note that it is not clear whether the MultiBlock improvement is of constant order or whether it reduces the complexity, and how it gains are affected by the value of the filter threshold

Silk 2.4.1 (MultiBlock, 8 cores)

This is the only experiment that used more than a single core: on 8 cores, Silk 2.4.1 finishes in 2 minutes and 30 seconds. This shows that the data integration algorithm is well parallelizable and indeed strongly benefits from the Map Reduce techniques implemented for Silk.

6.6 Conclusion

RDF Gears enables the implementation of the same algorithms as Silk does. Silk requires specifying the parameters and the Link Condition in the XML-based Silk LSL file. RDF Gears requires implementation in an RGL workflow. Although the Silk LSL file is a bit more concise and arguably easier to create, the example shows that incomplete results are generated when multiple directors are specified for a single movie. This implicit behavior illustrates that Silk still requires understanding of the internal algorithm to effectively interlink datasets. The RDF Gears workflow provides more transparency on what happens internally. Also, the genericity of RGL allows extending the algorithm to perform tasks not expressible by Silk.

The performance of RDF Gears was compared with that of Silk. The 2.0 version does not implement the MultiBlock query optimization algorithm and seems to have some performance problems. RGL offers fine control over the algorithm behavior, as shown by the elimination of the duplicate labels in the SPARQL query of the example. Also, the implementation is highly optimized and executes the query efficiently. This makes the RDF Gears workflow three times faster than Silk 2.0.

If a naive RDF Gears approach (where duplicate labels are not eliminated) is compared to a royally compensated version of Silk 2.0 (we ignore the time it spends on paging and the apparent 'sys' time bug), then Silk without MultiBlock would be 16% faster than RDF Gears. If an algorithm implemented in RDF Gears executes a mere 16% slower than an implementation in Java, the time saved with the high-level RGL language comes at very little cost.

The newer version of Silk 2.4.1 was used to see how parallelization and query optimization can improve the performance of data integration algorithms. Silk 2.4.1 applies MultiBlock, which makes that data integration twice as fast as RDF Gears. Also, when the algorithm is executed on multiple cores in parallel, runtime improves significantly.

The example algorithm benefits from the MultiBlock optimization and parallel execution, making the Silk implementation fast. Silk thus remains a useful and efficient tool for the specific task it is designed for. The big advantage of RDF Gears is that it provides more expressivity than Silk, while still providing an efficient implementation. It does not currently implement any parallelization or query optimization techniques, but the language is designed with such optimizations in mind. It will therefore be interesting to develop this for RDF Gears later.

Chapter 7

Conclusion and Future Work

This chapter concludes our research. Section 7.1 answers the research questions by summarizing our results: the RDF Gears Language, the user interface, and the engine implementation that we developed. Section 7.2 discusses possible future work, as there is room for improvements.

7.1 Summary

This thesis presented the design and implementation of the RDF Gears system. The design of the RGL language comprises the formal definition of the data model, syntax and semantics, and presents two concrete syntaxes: a graphical workflow-syntax and an XML format. The implementation is composed of the graphical user interface prototype to draw workflows, and the engine to execute the generated RGL expressions. The graphical user interface is a web application in which workflows can be created, stored and deleted, modified, shared and executed. The engine executes the workflows. It is designed and optimized to execute workflows with large amounts of data and contains hooks to allow extensibility by custom functions and secondary storage mechanisms.

I believe that these results justify the claim that it is indeed possible to implement a Semantic Web data integration framework that is expressive, useful and can be implemented efficiently. Below, the research questions are answered one by one.

1. *Is it possible to design a data transformation language and formally define its syntax and semantics?*

Chapter 3 presents the RDF Gears Language (RGL), showing that this is indeed possible. It formally defines the data model, which is based on the nested structures of the Named Nested Relational Calculus and is tailored to include RDF values (literals and URI's) and graphs. It defines the syntax of RGL, inspired by that of scientific workflow systems, as well as the RGL semantics. The RGL core operators implement the Nested Relational Algebra (NRA) and provide SPARQL query functionality. All these are specified unambiguously with set theoretic definitions.

RGL allows the addition of new functions to augment the expressivity. Examples of operators that are implemented for RDF Gears are string similarity

metrics and an operator that fetches Linked Data from the Web. Another custom operator that could be implemented is one to query relational databases.

2. ***What is the expressive power of that language?***

The expressive power has been assessed in a theoretical as well as in a practical evaluation. Section 3.7.2 contains a formal discussion of the RGL expressivity and shows that RGL is as expressive as the Nested Relational Algebra and Calculus. More specifically, it can at least express the language the language $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, -, \cdot, \div, \Sigma, =)$. This means the expressivity of a range of other nested relational languages from literature is covered.

A practical illustration of what RGL can do is the example of section 3.1, where an RGL workflow collects Linked Data. A more complex algorithm is given in the evaluation of chapter 6. Here it is shown how RGL can be used to implement the Silk algorithm, and how this algorithm can be modified to perform data integration in a way that Silk cannot.

3. ***What concrete syntax can be used for that language, and how can a user interface be implemented for it?***

Two concrete syntaxes are given. The first one is a graphical syntax that is convenient for users of RDF Gears. The syntax is inspired by Taverna and visualizes RGL workflows in a graphical way. It is used throughout the thesis to exemplify workflows, the most complex examples given in figures 3.1 and 6.1.

Chapter 4 presented a graphical user interface prototype implemented for RDF Gears. It is an AJAX Web application that allows the construction of workflows using Mozilla Firefox. It stores these in an XML format that is detailed in section 4. This XML format is also another concrete syntax for RGL, being better serializable than the graphical representation.

4. ***Is the language useful for scientists and developers using Semantic Web data?***

As stated in the beginning of this thesis, a full evaluation of usefulness is not within the scope of this thesis. Nevertheless, I believe that RGL can be said to be useful for the target users. Section 3.7.3 evaluated the usefulness of the RDF Gears Language, and illustrated that RGL meets the three properties we defined.

- a) *The language is expressive enough to express custom data integration algorithms:* the expressivity analysis of section 3.7.2 provides a theoretic assessment of the language possibilities. The LocalSPARQL and RemoteSPARQL operators provide operators that allow to easily construct SPARQL queries using intermediate results. The possibilities of RGL are further illustrated by the fact that a few operators can be combined to collect RDF data by following links in Linked Data (figure 3.1). Furthermore, the Silk equivalent workflow (figure 6.1) shows that RGL can be used to implement instance matching algorithms. The flexibility of the workflow paradigm allows to combine such functionalities in even larger workflows, performing more complex tasks.
- b) *The language provides a ‘right’ level of abstraction to make the expression of such algorithms convenient:* although this remains subjective and use case dependent, the examples strongly suggest that RGL indeed strikes a

balance between abstraction and expressivity. RGL is not as abstract and concise as a specialized language that is designed for one specific task (e.g. the LSL format for Silk), but as a result it has more expressivity such that it allows the implementation of new algorithms. RGL is far more abstract and therefore more concise than a general purpose programming language, thus allowing the user to create complex algorithms in little time. More operators and syntactic sugar can be added to RGL to further increase abstraction.

- c) *The functionality is clearly defined and understandable.* Apart from the fact that the language is formally defined, section 3.1 also shows that the semantics can be explained in plain English. The similarity of RGL with the Taverna language suggests that RGL can also be understood by non-programmers, and the example workflows show that it is not too difficult to implement data integration algorithms. It is therefore believed that indeed, this language is understandable for developers and scientists using Semantic Web data.

5. *How can an efficient execution engine for that language be implemented?*

Chapter 5 describes the implementation of the RDF Gears Engine, which executes workflows defined in the XML syntax. A static typechecking mechanism checks whether a workflow is executable and provides useful error messages if it is not. A query paging mechanism allows loading large batches of remote data from endpoints that limit the query result size. Query results are output in XML or in a more concise format useful for manual inspection.

The engine employs aggressive pipelining, which allows workflows to operate on large datastructures without imposing large memory requirements. Experimental results (figure 5.12) show that the pipelining mechanism should not be applied to all bags, and an optimizer is shown to be effective in determining when this pipelining can be safely applied as fire-and-forget pipelining, or when it is beneficial to store the results in memory. Lazy evaluation is implemented to postpone the evaluation of values until strictly necessary. It prevents unnecessary evaluations and is a saver in the absence of a sophisticated query rewriting algorithm, but also in cases where such an algorithm would not help because custom operators have nontransparent semantics. The static typing of the language allows implementation of a technique we call Field Map Indexing, which provides a constant order reduction of up to 60% for both memory and run time (figure 5.13).

These and many constant order optimizations in the RGL implementation make that the engine is fast. Even compared with a procedurally implemented special purpose algorithm, RDF Gears is marginally slower than Silk 2.0 and a few times slower than Silk 2.4 that implements a sophisticated special purpose optimization technique (section 6.5).

There is still room for improvement. Section 3.7.4 showed how the design of RGL is suitable for the implementation of query optimization techniques that are known in literature, such as the query rewriting techniques of the Nested Relational Algebra. Implementations that parallelize workflow execution, for example with the Map Reduce framework, are also likely to be implementable.

7.2 Future work

The results of this research are hoped to be a contribution to the Web Information Systems group. However, there is much to do still. Future work can be categorized as work on the language, on the user interface, or on the engine that executes the language.

Research on the RGL language, and in particular the relation between NRC and SPARQL, may provide more optimization rules that allow static query rewriting. If these are implemented for RGL and the engine reorders the query before executing it, it may save a lot of time. For example, a filter that is pushed forward will reduce the intermediate results and save unnecessary work.

New operators may be invented for the RGL language. For example, some operators will provide new functionality, such as an operator that queries a relational database or string-manipulation operators. Other operators may provide a syntactic sugar that now requires cumbersome simulation in NRC, such as the construction of a bag of n predefined elements. New operators that would be added to the RGL core operators should be carefully considered as the optimizer should be able to reorder them.

By extending the semantics of RGL-SPARQL to allow a *set* of values to be bound to a variable (instead of a *single* value), RGL can possibly even be integrated with the SPARQL federation extensions [39] in the future to perform distributed semi-joins.

The user interface is currently a prototype and its usability can be improved. First of all, it should be extended to visualize the workflows on more than 1 level: the nested workflows should be transparently visible and editable in a reusing workflow. Also, typechecking help is currently provided by the engine before workflow execution, but it would be nice to display this real-time in the workflow, while drawing it. Two bugs in the used libraries should be fixed. The collaborative features are a nice proof-of-concept, but serious collaboration requires functionalities like user login, selective sharing of workflows, workflow searching. For all user interface issues it is well worth to first study Taverna in more detail, as a lot of experience with usability is concentrated in the product.

The integration of the engine and user interface can be improved. The user interface currently exposes a HTTP URI that, upon resolving, executes the workflows and provides the results. This way it is a proof-of-concept of how to publish Linked Data, but it will not scale well. Therefore it is often more convenient to run the engine separately. Also, the HTTP interface may be improved, e.g. by allowing the specification of workflow input arguments.

The execution engine is hoped to be much more than a prototype, as great care was taken to design and implement it such that it is flexible and extensible. It would be useful to execute more performance measurements, in a wider variety of usecases, for the engine. It may well benefit from further optimization. The pipelining optimizer may be refined. Also, a secondary storage mechanism may be implemented to allow materialization of large intermediate results. A query rewriting mechanism seems useful; it will mainly require a study of the RGL language, not so much changing the engine architecture. Other optimization techniques such as query executing planning using cost-estimation techniques and mapping RGL queries to the Map Reduce framework may furthermore improve the performance, although they may also require more

rigorous changes to the implementation.

Finally, RDF Gears can be improved with facilities to debug workflows, provide an estimation of execution progress, inspect intermediate results and to save and reload these.

Bibliography

- [1] Linked Data Design Issues. <http://www.w3.org/DesignIssues/LinkedData.html>.
- [2] "Silk querying mechanism". Personal communication with Robert Isele (Silk). Silk mailing list archives, 17 July 2011.
- [3] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumüller. Triplify: light-weight linked data publication from relational databases. In *Proceedings of the 18th international conference on World wide web*, pages 621–630. ACM New York, NY, USA, 2009.
- [4] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. DBpedia: A Nucleus for a Web of Open Data. *Lecture Notes in Computer Science*, 4825:722–735, 2007.
- [5] T. Berners-Lee and J. Hendler. Scientific publishing on the semantic web. *Nature*, 410:1023–1024, 2001.
- [6] C. Bizer and R. Cyganiak. D2R Server—publishing relational databases on the semantic web. In *Poster at the 5th International Semantic Web Conference*. Cite-seer, 2006.
- [7] Chris Bizer, Richard Cyganiak, Jörg Garbers, Oliver Maesh, and Christian Becker. The D2RQ Platform v0.7 - Treating Non-RDF Relational Databases as Virtual RDF Graphs. <http://www4.wiwiwiss.fu-berlin.de/bizer/d2rq/spec/20090810/>.
- [8] P. Buneman, R. E. Frankel, and R. Nikhil. An implementation technique for database query languages. *ACM Transactions on Database Systems (TODS)*, 7(2):164–186, 1982.
- [9] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.
- [10] J. V. Bussche and S. Vansummeren. Polymorphic type inference for the named nested relational calculus. *ACM Transactions on Computational Logic (TOCL)*, 9(1):3, 2007.

-
- [11] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. What is query rewriting? *Cooperative Information Agents IV-The Future of Information Agents in Cyberspace*, pages 439–457, 2004.
- [12] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.
- [13] B. Chen, X. Dong, D. Jiao, H. Wang, Q. Zhu, Y. Ding, and D. Wild. Chem2Bio2RDF: a semantic framework for linking and data mining chemogenomic and systems chemical biology data. *BMC bioinformatics*, 11(1):255, 2010.
- [14] J. Chen and Q. Huang. Eliminating the impedance mismatch between relational systems and object-oriented programming languages. *Monash University, Australia*, 56, 1995.
- [15] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the IJCAI-2003 Workshop on Information Integration on the Web (IIWeb-03)*, pages 73–78. Citeseer, 2003.
- [16] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [17] R. Dhamankar, Y. Lee, A. H. Doan, A. Halevy, and P. Domingos. iMAP: discovering complex semantic matches between database schemas. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 383–394. ACM New York, NY, USA, 2004.
- [18] A. H. Doan and A. Y. Halevy. Semantic integration research in the database community: A brief survey. *AI magazine*, 26(1):83, 2005.
- [19] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.
- [20] Steve H. Garlik, Andy Seaborne, and Eric Prud’hommeaux. SPARQL 1.1 Query Language. <http://www.w3.org/TR/sparql11-query/>.
- [21] A. Halevy, A. Rajaraman, and J. Ordille. Data integration: The teenage years. In *Proceedings of the 32nd international conference on Very large data bases*, page 16. VLDB Endowment, 2006.
- [22] Olaf Hartig, Christian Bizer, and Johann C. Freytag. Executing SPARQL Queries over the Web of Linked Data. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *International Semantic Web Conference*, volume 5823 of *Lecture Notes in Computer Science*, pages 293–309. Springer, 2009.
- [23] O. Hassanzadeh and M. Consens. Linked movie data base. In *Proceedings of the 2nd Workshop on Linked Data on the Web (LDOW2009)*, 2009.

- [24] O. Hassanzadeh, R. Xin, R. J. Miller, A. Kementsietsidis, L. Lim, and M. Wang. Linkage query writer. *Proceedings of the VLDB Endowment*, 2(2):1590–1593, 2009.
- [25] M. L. Hetland. Divide, Combine, and Conquer. *Python Algorithms: Mastering Basic Algorithms in the Python Language*, pages 125–150, 2011.
- [26] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. Van den Bussche. A formal model of dataflow repositories. In *Proceedings of the 4th international conference on Data integration in the life sciences*, pages 105–121. Springer-Verlag, 2007.
- [27] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic acids research*, 34(suppl 2), 2006.
- [28] R. Isele, A. Jentzsch, and C. Bizer. Efficient Multidimensional Blocking for Link Discovery without losing Recall. June 2011.
- [29] A. Jentzsch, R. Isele, and C. Bizer. Silk-Generating RDF Links while publishing or consuming Linked Data. In *Poster at the International Semantic Web Conference (ISWC2010), Shanghai*, 2010.
- [30] Y. Kalfoglou and M. Schorlemmer. Ontology mapping: the state of the art. *The knowledge engineering review*, 18(01):1–31, 2003.
- [31] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. <http://www.w3.org/TR/rdf-concepts/>.
- [32] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.
- [33] M. Laclavik. RDB2Onto: Relational Database Data to Ontology Individuals Mapping. *Tools for Acquisition, Organisation and Presenting of Information and Knowledge*, pages 86–89, 2006.
- [34] M. Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246. ACM New York, NY, USA, 2002.
- [35] B. Lud"ascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.
- [36] C. Morbidoni, D. Le Phuoc, A. Polleres, M. Samwald, and G. Tummarello. Pre-viewing semantic web pipes. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, pages 843–848. Springer-Verlag, 2008.

-
- [37] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, and Others. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.
- [38] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.
- [39] Eric Prud'hommeaux and Andy Seaborne. SPARQL 1.1 Federation Extensions. <http://www.w3.org/TR/sparql11-federated-query/>.
- [40] B. Quilitz and U. Leser. Querying distributed rdf data sources with sparql. *Lecture Notes in Computer Science*, 5021:524, 2008.
- [41] P. Ravindra, V. V. Deshpande, and K. Anyanwu. Towards scalable RDF graph analytics on MapReduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, page 5. ACM, 2010.
- [42] S. S. Sahoo, O. Bodenreider, J. L. Rutter, K. J. Skinner, and A. P. Sheth. An ontology-driven semantic mashup of gene and biological pathway information: Application to the domain of nicotine dependence. *Journal of biomedical informatics*, 41(5):752–765, 2008.
- [43] M. Salvadores, G. Correndo, B. Rodriguez-Castro, N. Gibbins, J. Darlington, and N. Shadbolt. LinksB2N: Automatic Data Integration for the Semantic Web. In Robert Meersman, Tharam S. Dillon, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2009*, volume 5871, pages 1121–1138. Springer, 2009.
- [44] Simon Schenk and Steffen Staab. Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the web. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 585–594, New York, NY, USA, 2008. ACM.
- [45] B. Schoenmakers. Designing Linked Data Applications. Master's thesis, 2010.
- [46] Andy Seaborne. response to "Bug in com.hp.hpl.jena.sparql.expr.NodeValue". <http://tech.groups.yahoo.com/group/jena-dev/message/46389>.
- [47] Andy Seaborne. response to "How to see remote SPARQL execution errors?". <http://tech.groups.yahoo.com/group/jena-dev/message/47656>.
- [48] J. F. Sequeda, C. Cunningham, R. Depena, and D. P. Miranker. Ultrawrap: Using SQL Views for RDB2RDF. In *8th International Semantic Web Conference*.
- [49] R. Sridhar, P. Ravindra, and K. Anyanwu. RAPID: Enabling scalable ad-hoc analytics on the semantic web. *ISWC 2009*, pages 715–730, 2009.
- [50] J. Sroka, J. Hidders, P. Missier, and C. Goble. A formal semantics for the Taverna 2 workflow model. *Journal of Computer and System Sciences*, 76(6):490–508, 2010.

BIBLIOGRAPHY

- [51] A. Thor and E. Rahm. CloudFuice: A Flexible Cloud-Based Data Integration System. *Web Engineering*, pages 304–318, 2011.
- [52] Andreas Thor and Erhard Rahm. CloudFuice: A flexible Cloud-based Data Integration System. Technical report, University of Leipzig, April 2011.
- [53] TopQuadrant. SPARQLMotion. <http://topquadrant.com/products/SPARQLMotion.html>.
- [54] H. Wache, T. Voegelé, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. Ontology-based integration of information-a survey of existing approaches. In *IJCAI-01 Workshop: Ontologies and Information Sharing*, volume 2001, pages 108–117. Citeseer, 2001.
- [55] L. Wong. *Querying nested collections*. PhD thesis, University of Pennsylvania, 1994.

Appendix A

Report of an earlier RGL design attempt

When designing the RDF Gears Language, I initially set out with as only data type an *annotated graph*. This is a tuple (G, V) with G an RDF graph and V a set of *marked nodes* (URI's or literals that may or may not be used in G). Thus the marked nodes would represent the item upon which a function could operate, and the graph could provide extra data about the item.

Because we first wanted to research expressivity, not usability, I showed that plain RDF graphs could be used to simulate these annotated graphs¹ and also the SPARQL solution sequences that resulted from SELECT queries. I furthermore observed that the SPARQL SELECT queries themselves could be simulated with SPARQL CONSTRUCT and thus offered no extra expressivity (except possibly for constructs like LIMIT and ORDER). Thus, a transformation of a number of input graphs to a single output graph could be implemented with a SPARQL CONSTRUCT (or imperatively, if not expressible in SPARQL). A combination of such functions a workflow would allow for complex data transformations.

Then the traditional NRC was selected merely as a benchmark for expressivity. I created a way to simulate an NRC datastructure in RDF, and simulated the basic NRC operations with SPARQL CONSTRUCT queries over the input graphs (e.g. the pair formation function operates on two inputs, and the equivalent CONSTRUCT query would thus query two graphs). This led to some interesting observations. Firstly, that the use of blank nodes in the simulation of NRC data structures prohibits reference to those structures from outside the graph, as the semantics of a blank node identifier is only defined within its own graph or solution sequence. Secondly, that we needed the SPARQL 1.1 BIND function to apply some tricks necessary to simulate the tensor function of traditional NRC (the tensor function is comparable to the Cartesian product). Finally it turned out to be possible to simulate the basic NRC constructs with SPARQL CONSTRUCT, at least if we did not require parametric polymorphism. Thus simulating NRC with a workflow of SPARQL CONSTRUCT queries was possible, but very cumbersome.

Finally, experimentation with many mockups of a graphical syntax led to the observation that although the language was as expressive as NRC, it would not be very

¹By reserving some URI x and extending G with a triple (v, x, v) for every $v \in V$

easy to use. Thus we experimented with various extensions of the data model (allowing records, sets of records, etcetera), until we decided to simply adopt the type system of the Named Nested Relational Calculus, with Graphs and RDF atoms (i.e. URI's and literals) as base types.

Appendix B

RGL XML syntax

The format of the XML syntax for RGL workflow specification files is defined by the following Document Type Definition.

```
<!DOCTYPE rdfgears [  
<!ELEMENT rdfgears (metadata, workflow)>  
  <!-- metadata contains some info about this workflow -->  
  <!ELEMENT metadata (id,name,password?)>  
    <!-- the workflow id must correspond to the workflow xml path+filename.  
         Here the path is relative to a workflow path, and filename excludes  
         the .xml extension -->  
    <!ELEMENT id (#PCDATA)>  
    <!ELEMENT description (#PCDATA)>  
    <!ELEMENT password (#PCDATA)>  
  
  <!-- the workflow definition itself -->  
  <!ELEMENT workflow (workflowInputList, network)>  
    <!ELEMENT workflowInputList (workflowInputPort)>  
    <!ATTLIST workflowInputList  
      x CDATA #IMPLIED  
      y CDATA #IMPLIED>  
  
    <!ATTLIST workflowInputPort  
      name CDATA #REQUIRED>  
  
    <!ELEMENT workflowInputPort (#PCDATA)>  
    <!-- the networks output-attribute specifies what processor generates the  
         workflow output -->  
    <!ELEMENT network (processor+)>  
    <!ATTLIST network  
      output CDATA #REQUIRED  
      x CDATA #IMPLIED  
      y CDATA #IMPLIED>  
  
    <!-- the processor x,y coordinates are optional, and only used for  
         visualisation in the workflow-->  
    <!ELEMENT processor (function, inputPort*)>  
    <!ATTLIST processor  
      id ID #REQUIRED  
      x CDATA #IMPLIED  
      y CDATA #IMPLIED>  
    <!-- if a function's type-attribute is "custom-java", it must have a
```

```

<config param="implementation"></config> whose PCDATA specifies
the implementing java class .
For these and other functions, all config parameters specified are
passed to the rglFunction.initialize() function after instantiating
the RGLFunction -->
<!--ELEMENT function (config*)>
<!--ATTLIST function
    type (workflow|custom-java|constant|comparator|filter|sparql|if|
        record-project|record-create|record-union|record-join|
        bag-flatten|bag-singleton|bag-union) #REQUIRED>

    <!--ELEMENT config (#PCDATA)>
    <!--ATTLIST config
        param CDATA #REQUIRED>
<!--ELEMENT inputPort (source)>
    <!--ATTLIST inputPort
        iterate (true|false) "false"
        name CDATA #REQUIRED>
    <!-- A source refers to the element generating the input for this port.
        it contains either a 'processor' attribute with a
        processor id, or a workflowInputPort attribute with the name
        of a <workflowInputPort> element -->
    <!--ELEMENT source (#PCDATA)>
    <!--ATTLIST source
        processor IDREF #IMPLIED
        workflowInputPort CDATA #IMPLIED>

```

```

]>

```

Appendix C

Performance measurement data

All experiments were conducted on a 64 bit system with an 8 core Intel Xeon CPU and 16GB of memory running GNU/Linux. Java version was 1.6.0. Note that RDF Gears is a single threaded application and was thus executed on a single core.

C.1 Evaluation of the optimizer

The configuration of RDF Gears was as follows in all three experiments:

- the RDF Gears config file contained `greedy_loading_of_remote_queries = false`
- Lazy evaluation was disabled for all functions
- Field Index Mapping was disabled
- Java was run without assertions (in `rdfgears` script)

The workflow `testStream` (figure 5.11) was executed with the command

```
$ ./rdfgears -w testStream -d debug --outputformat none
```

The output format 'none' evaluates all results but does not incur the overhead of string serialization.

We used the `-Xmx` flag for the java command to vary the maximum heap value available to the JVM. In every experiment we set the heap size limit low enough to force regular garbage collection (thus reducing the measurement uncertainty of the actually required heap size) but not so high that the cost of garbage collection would become significant in processing time (see figure C.1).

C.1.1 Materializing results

In the materializing experiment RDF Gears was configured with `rememberResults=true` in the implementation of `StreamingBagValue` (this makes all streaming bags materializing).

Results:

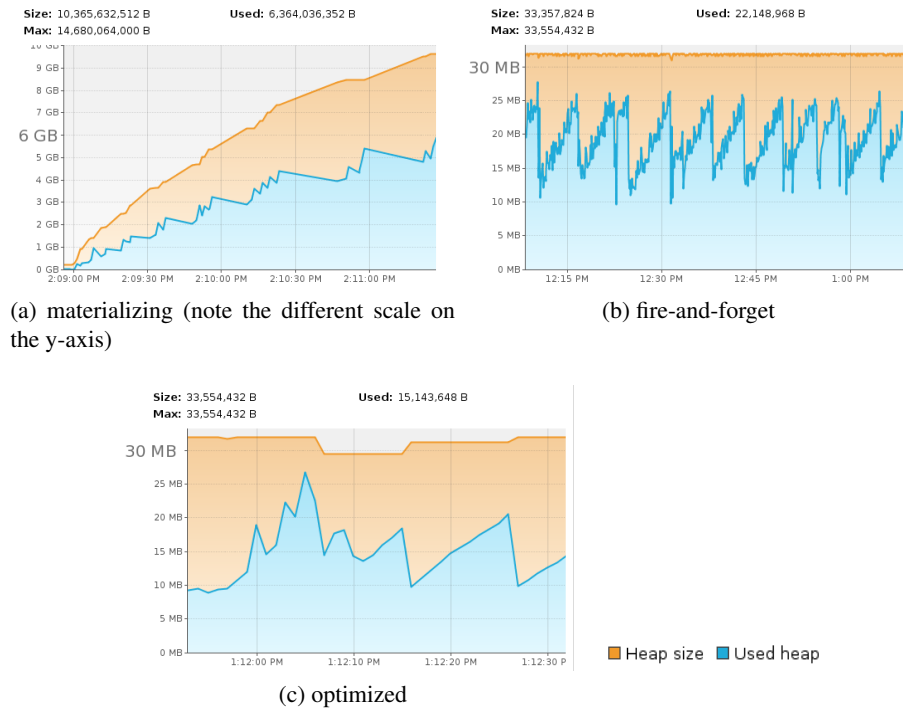


Figure C.1: Memory usage during workflow execution

<u>-Xmx</u>	<u> B </u>	<u> A . B </u> (millions)	<u>max used</u> <u>heap (MB)</u>	<u>time real</u>	<u>time user</u>	<u>time sys</u>
32	0	0	10.1	0m3.297s	0m2.799s	0m0.122s
32	0	0	10.1	0m3.649s	0m3.030s	0m0.119s
32	0	0	10.1	0m3.303s	0m2.846s	0m0.119s
1500	2000	5	fail			
3000	2000	5	1522	0m29.991s	1m7.177s	0m2.278s
3000	2000	5	1411	0m28.716s	0m58.677s	0m2.323s
3000	2000	5	1413	0m35.461s	1m14.945s	0m2.545s
3500	4000	10	fail			
7000	4000	10	3035	0m58.617s	2m6.423s	0m4.465s
7000	4000	10	3199	1m2.773s	2m33.301s	0m4.947s
7000	4000	10	3047	0m58.650s	2m7.830s	0m4.655s
5000	6000	15	fail			
10000	6000	15	4792	1m57.253s	4m43.183s	0m8.126s
10000	6000	15	4872	1m55.797s	4m33.154s	0m7.866s
10000	6000	15	4620	1m55.981s	4m38.341s	0m7.699s
7000	8000	20	fail			
14000	8000	20	6063	2m43.618s	7m6.096s	0m10.138s
14000	8000	20	5999	2m37.648s	6m22.933s	0m9.886s
14000	8000	20	6105	2m34.684s	6m0.501s	0m11.011s

Note:

- 'fail' means the JVM terminated due to a lack of available memory.
- the 'user' time exceeds the 'real' time (given by the GNU `time` command), indicating multiple active cores. This is not RDF Gears, but the multithreaded JVM implementation. As this does not happen with the other setups, we suspect this is the Java garbage collector and/or the memory allocation system that requires

significant CPU capacity to manage such large amounts of memory. This cost was not taken into account in the analysis.

C.1.2 Streaming results

In the streaming experiment RDF Gears was configured with `rememberResults=false` in the implementation of `StreamingBagValue` (this is the RDF Gears default). Furthermore the flag `--disable-optimizer` was appended to the `rdfgears` command.

Results:

<u>-Xmx</u>	<u> B </u>	<u> A . B </u> (millions)	<u>max used</u> <u>heap (MB)</u>	<u>time real</u>	<u>time user</u>	<u>time sys</u>
32	0	0	10.1	0m4.172s	0m3.029s	0m0.134s
32	0	0	10	0m3.595s	0m3.009s	0m0.124s
32	0	0	10	0m3.611s	0m3.043s	0m0.117s
32	2000	5	28.5	20m35.202s	4m58.167s	6m57.461s
32	2000	5	27.5	20m32.705s	4m41.808s	7m2.410s
32	2000	5	28.3	20m31.107s	4m37.687s	6m55.015s
32	4000	10	28	41m1.465s	9m21.375s	13m47.818s
32	4000	10	28.4	41m0.290s	9m19.397s	13m59.648s
32	4000	10	28.5	41m17.416s	9m15.394s	14m2.267s
32	6000	15	28.3	61m2.682s	13m36.675s	20m52.937s
32	6000	15	28.7	61m20.638s	14m4.139s	20m55.433s
32	6000	15	27.8	61m11.669s	13m44.248s	20m55.008s
32	8000	20	28.4	83m40.538s	17m53.862s	27m46.220s
32	8000	20	27.2	81m54.535s	18m49.484s	27m46.068s
32	8000	20	27.5	81m29.522s	20m11.940s	27m39.285s

C.1.3 Optimized results

In the streaming experiment RDF Gears was configured with `rememberResults=false` in the implementation of `StreamingBagValue` (this is the RDF Gears default).

Results:

<u>-Xmx</u>	<u> B </u>	<u> A . B </u> (millions)	<u>max used</u> <u>heap (MB)</u>	<u>time real</u>	<u>time user</u>	<u>time sys</u>
32	0	0	10.1	0m4.067s	0m2.865s	0m0.109s
32	0	0	10.1	0m3.608s	0m3.003s	0m0.128s
32	0	0	10.1	0m3.601s	0m3.028s	0m0.141s
32	2000	5	17.1	0m13.490s	0m14.543s	0m0.410s
32	2000	5	17.5	0m13.824s	0m14.603s	0m0.416s
32	2000	5	19.8	0m13.887s	0m14.802s	0m0.434s
32	4000	10	22.5	0m22.923s	0m24.577s	0m0.685s
32	4000	10	25.3	0m23.495s	0m25.283s	0m0.701s
32	4000	10	22	0m23.453s	0m24.984s	0m0.745s
32	6000	15	24.2	0m33.041s	0m35.208s	0m1.055s
32	6000	15	21.5	0m33.227s	0m35.092s	0m1.296s
32	6000	15	23.3	0m32.951s	0m35.237s	0m0.981s
32	8000	20	22.9	0m42.195s	0m44.361s	0m1.199s
32	8000	20	23.2	0m43.149s	0m45.726s	0m1.286s
32	8000	20	22.2	0m42.438s	0m44.658s	0m1.357s

C.2 Evaluation of Field Index Mapping

The experiments forced materialization of bags with `rememberResults=true` in the implementation of `StreamingBagValue` and executed the workflow `testStream`

(figure 5.11) with $|B| = 8000$.

Although bag materialization is not needed for all bags in the workflow, it is a good way to measure size of materialized bags with and without the Field Index Mapping technique.

All experiments were run with the command

```
$ ./rdfgears -w testStream -d debug --outputformat none
```

The output format 'none' evaluates all results but does not incur the overhead of string serialization.

C.2.1 No FieldIndex

Without FieldIndexing, each ValueRow instance stores its values in a `HashMap<String, RGLValue>` array. This setup was used for the pipelining evaluation, and the results are taken from that experiment (materializing, $|B| = 8000$) yielding the following average:

experiment	used heap size (MB)	time real	time user	time sys
average	6055.67	2m38.64s		

C.2.2 Field Index Mapping (HashMap)

For this experiment the `nl.tudelft.rdfgears.util.row.FieldIndexMapFactory` was modified to return instance of the class `nl.tudelft.rdfgears.util.row.FieldIndexHashMap`.

Three runs gave the following result:

experiment	used heap size (MB)	time real	time user	time sys
1	2415	0m56.097s	1m51.495s	0m4.165s
2	2302	0m54.032s	1m43.485s	0m4.088s
3	2277	0m55.336s	2m15.036s	0m3.797s
average	2331.33	0m55.15s		

C.2.3 Field Index Mapping (array)

For this experiment the `nl.tudelft.rdfgears.util.row.FieldIndexMapFactory` was modified to return instance of the class `nl.tudelft.rdfgears.util.row.FieldIndexArrayMap`.

Three runs gave the following result:

experiment	used heap size (MB)	time real	time user	time sys
1	2251	0m57.200s	1m53.345s	0m3.948s
2	1998	1m6.494s	2m13.649s	0m4.577s
3	2161	1m8.273s	2m25.818s	0m4.177s
average	2136.67	1m03.99s		

C.3 Performance comparison with Silk

The experiments executed the algorithm described by the Silk LSL file of figure C.2. RDF Gears used the workflow of figure 6.1. The links created by RDF Gears and Silk 2.0 have no difference in recall/precision, although for movies with multiple directors they may create a different link if both director-links have an identical similarity (see page 75).

Experiments were conducted with 100 MB of available heap space to the JVM and all RDF Gears engine optimizations enabled.

C.3.1 RDF Gears workflow

Executing the workflow of figure 6.1 without query paging.

```
$ time ./rdfgears -w workflows/silkExample/mainRDF-all -d ALL > results.xml
```

experiment	time real	time user	time sys
1	real 19m38.375s	user 19m51.287s	sys 0m12.217s
2	real 19m11.159s	user 19m37.175s	sys 0m12.876s

In a more naive approach, the workflow was configured as figure 6.1 but with the DBpedia query not applying `(str(?label_lang) AS ?label)` in the SPARQL query. Results:

experiment	time real	time user	time sys
1	real 41m46.497s	user 42m54.021s	sys 0m25.346s
2	real 40m49.556s	user 41m41.330s	sys 0m24.416s

With query paging enabled (1000 results per page, thus 186 queries), the results are:

experiment	time real	time user	time sys	query time
1	real 49m2.003s	user 42m27.503s	sys 0m24.941s	8m51s
2	real 49m3.507s	user 42m0.620s	sys 0m25.388s	8m42s
3	real 48m11.772s	user 41m12.677s	sys 0m25.737s	8m04s

C.3.2 Silk 2.0 (without MultiBlock)

Silk 2.0 was used to measure the matching time without the use of the MultiBlock method. Silk 2.0 applies the paging mechanism of section (5.4.3) with a fixed page size of 1000. Silk 2.0 thus used 189 queries to load 186.956 movie-label tuples.

Silk was executed with the command

```
$ time java -Dthreads=1 -DconfigFile=examples/linkedmdb_directors.xml -DlogQueries=true -jar silk.jar
```

Results:

experiment	time real	time user	time sys	query time
1	66m16.673s	36m29.998s	22m50.117s	8m50.667s
2	66m36.727s	37m24.338s	23m25.076s	8m24.111s

C.3.3 Silk 2.4.1 (with MultiBlock)

Silk 2.4.1 was used to run Silk with the MultiBlock method. It was ran with the command

```
$ time java -Dthreads=1 -DconfigFile=examples/linkedmdb_directors.xml -DlogQueries=true -jar silk.jar
```

Results:

experiment	time real	time user	time sys
1	11m59.753s	12m4.402s	0m18.697s
2	11m16.175s	11m23.773s	0m18.511s

It must be noted that this did actually not return the same links as Silk 2.0 and RDF Gears. This is caused by a change in the Silk LSL format. Silk 2.4.1 does still execute the old LSL file, although with a warning. I have no reason to believe that the execution time is positively affected by this, as all links do have a calculated similarity. As I trust that Silk 2.4.1 will also be capable of generating the correct results if the LSL file is properly modified, I consider the measurements valid.

In multithreaded execution it takes as much computation power (user time), but large parts of the algorithm can be executed on multiple cores, reducing the runtime. The command

```
time java -Dthreads=8 -DconfigFile=examples/linkedmdb_directors.xml -DlogQueries=true -jar silk.jar
```

executes Silk on 8 cores with the following results:

experiment	time real	time user	time sys
1	2m24.673s	12m14.312s	0m18.674s
2	2m26.839s	12m13.442s	0m19.417s
3	2m26.265s	12m24.067s	0m18.965s

```

<?xml version="1.0" encoding="utf-8" ?>
<Silk>
  <Prefixes>
    <Prefix id="rdf" namespace="http://www.w3.org/1999/02/22-rdf-syntax-ns#" />
    <Prefix id="rdfs" namespace="http://www.w3.org/2000/01/rdf-schema#" />
    <Prefix id="dbpedia" namespace="http://dbpedia.org/ontology/" />
    <Prefix id="movie" namespace="http://data.linkedmdb.org/resource/movie/" />
  </Prefixes>

  <DataSources>
    <DataSource id="dbpedia" type="sparqlEndpoint">
      <Param name="endpointURI" value="http://localhost:2020/sparql" /> <!-- DBPedia -->
      <Param name="graph" value="file:Data/dbpedia.n3" />
      <Param name="pageSize" value="500000" /> <!-- silk 2.4 only -->
    </DataSource>
    <DataSource id="linkedmdb" type="sparqlEndpoint">
      <Param name="endpointURI" value="http://localhost:2020/sparql" /> <!-- LinkedMDB -->
      <Param name="graph" value="file:Data/lmdb.n3" />
      <Param name="pageSize" value="500000" /> <!-- silk 2.4 only -->
    </DataSource>
  </DataSources>

  <Interlinks>
    <Interlink id="movies">
      <LinkType>dbpedia:director</LinkType>

      <SourceDataset dataSource="dbpedia" var="a">
        <RestrictTo>
          ?a rdf:type dbpedia:Film
        </RestrictTo>
      </SourceDataset>
      <TargetDataset dataSource="linkedmdb" var="b">
        <RestrictTo>
          ?b rdf:type movie:director
        </RestrictTo>
      </TargetDataset>

      <LinkCondition>
        <Aggregate type="max">
          <Compare metric="jaro">
            <Input path="?a/dbpedia:director/rdfs:label" />
            <Input path="?b/rdfs:label" />
          </Compare>
          <Compare metric="jaro">
            <Input path="?a/dbpedia:director/rdfs:label" />
            <Input path="?b/movie:director_name" />
          </Compare>
        </Aggregate>
      </LinkCondition>

      <Filter threshold="0.7" limit="1" />

      <Outputs>
        <Output maxConfidence="0.9" type="file">
          <Param name="file" value="lmdb_verify_links.xml"/>
          <Param name="format" value="alignment"/>
        </Output>
        <Output minConfidence="0.9" type="file">
          <Param name="file" value="lmdb_accepted_links.xml"/>
          <Param name="format" value="alignment"/>
        </Output>
      </Outputs>
    </Interlink>
  </Interlinks>
</Silk>

```

Figure C.2: Silk LSL file