LarKC

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

FP7 – 215535

D4.4.1 Investigation & Design for Rule-based Reasoning

Coordinator: Florian Fischer (STI Innsbruck)
With contributions from: Vassil Momtchev (OntoText)
Quality Assessor: Luka Bradesko (CycEur)
Quality Controller: Zhisheng Huang (VUA)

Document Identifier:	LarKC/2008/D4.4.1/Vx.x
Class Deliverable:	LarKC EU-IST-2008-215535
Version:	version 1.0
Date:	June 25, 2010
State:	final
Distribution:	public

Executive Summary

Rule-based reasoning and techniques offer viable solutions to the challenges posed by Web-scale reasoning. Rule-engines can be used to reason with multiple tractable fragments of standard formalisms, and rules themselves offer a convenient modeling approach to many domain specific problems.

In this document, we discuss methods to adapt existing rule-based inference techniques in order to meet LarKC's goals of reasoning at a Web-scale by employing approximation methods and/or parallelism. After an initial examination of the role of rule-based reasoning on the Web, and therefore also for LarKC, we analyze methods and approaches that utilize parallel architectures, taking in to account issues such as the problem of data-partitioning and -distribution that play central roles for the parallel evaluation of rules. Based on this, we describe a concrete extensions, based on Map-Reduce, to an existing rule-engine (IRIS), which allows the parallel computation of joins during the evaluation of individual rules.

Combined with standardization results stemming from the RIF working group this allows to employ IRIS as a scalable, parallel inference engine for generic RIF-Core rule-sets, which indirectly also facilitates parallel OWL 2 RL and RDFS reasoning.

Document Information

IST Project Number	FP7 – 215535	Acronym	LarKC
Full Title	The Large Knowledge Collider: a platform for large scale integrated reasoning and Web-search		
Project URL	`http://www.larkc.eu/`		
Document URL			
EU Project Officer	Stefano Bertolo		

Deliverable	Number	4.4.1	Title	Investigation & Design for Rule-based Reasoning
Work Package	Number	4	Title	Reasoning and Deciding

Date of Delivery	Contractual	M27	Actual	31-Jun-10
Status	version 1.0		final ☒	
Nature	prototype ☐ report ☒ dissemination ☐			
Dissemination Level	public ☒ consortium ☐			

Authors (Partner)	Vassil Momtchev (OntoText)			
Resp. Author	Florian Fischer		E-mail	florian.fischer@sti2.at
	Partner	STI Innsbruck	Phone	+43 512 507 6426

Abstract (for dissemination)	Rule-based reasoning and techniques offer viable solutions to the challenges posed by Web-scale reasoning. Rule-engines can be used to reason with multiple tractable fragments of standard formalisms, and rules themselves offer a convenient modeling approach to many domain specific problems. In this document, we discuss methods to adapt existing rule-based inference techniques in order to meet LarKC's goals of reasoning at a Web-scale by employing approximation methods and/or parallelism. After an initial examination of the role of rule-based reasoning on the Web, and therefore also for LarKC, we analyze methods and approaches that utilize parallel architectures, taking in to account issues such as the problem of data-partitioning and -distribution that play central roles for the parallel evaluation of rules. Based on this, we describe a concrete extensions, based on Map-Reduce, to an existing rule-engine (IRIS), which allows the parallelization of join computations during the evaluation of individual rules. Combined with standardization results stemming from the RIF working group this allows to employ IRIS as a scalable, parallel inference engine for generic RIF-Core rule-sets, which indirectly also facilitates parallel OWL 2 RL and RDFS reasoning.

Keywords	rules, reasoning, RIF, parallelization

Version Log			
Issue Date	**Rev No.**	**Author**	**Change**
18.02.2010	1	Florian Fischer	Document creation and inital sections
03.04.2010	2	Florian Fischer	Background, relevance for Web-scale reasoning and LarKC
28.05.2010	3	Florian Fischer	Parallelization and Reasoner Architecture
24.05.2010	4	Florian Fischer	Changes according to Reviewer Comments

Project Consortium Information

Participant's name	Partner	Contact
Semantic Technology Institute Innsbruck, Universitaet Innsbruck		Dieter Fensel Semantic Technology Institute (STI), Universitaet Innsbruck, Innsbruck, Austria Email: dieter.fensel@sti-innsbruck.at
AstraZeneca AB		Bosse Andersson AstraZeneca Lund, Sweden Email: bo.h.andersson@astrazeneca.com
CEFRIEL - SOCIETA CONSORTILE A RESPONSABILITA LIMITATA		Emanuele Della Valle CEFRIEL - SOCIETA CONSORTILE A RESPONSABILITA LIMITATA Milano, Italy Email: emanuele.dellavalle@cefriel.it
CYCORP, RAZISKOVANJE IN EKSPERIMENTALNI RAZVOJ D.O.O.		Michael Witbrock CYCORP, RAZISKOVANJE IN EKSPERIMENTALNI RAZVOJ D.O.O., Ljubljana, Slovenia Email: witbrock@cyc.com
Höchstleistungsrechenzentrum, Universitaet Stuttgart		Georgina Gallizo Höchstleistungsrechenzentrum, Universitaet Stuttgart Stuttgart, Germany Email : gallizo@hlrs.de
MAX-PLANCK GESELLSCHAFT ZUR FOERDERUNG DER WISSENSCHAFTEN E.V.		Lael Schooler, Max-Planck-Institut für Bildungsforschung Berlin, Germany Email: schooler@mpib-berlin.mpg.de
Ontotext AD		Atanas Kiryakov, Ontotext Lab, Sofia, Bulgaria Email: naso@ontotext.com
SALTLUX INC.		Kono Kim SALTLUX INC Seoul, Korea Email: kono@saltlux.com
SIEMENS AKTIENGESELLSCHAFT		Volker Tresp SIEMENS AKTIENGESELLSCHAFT Muenchen, Germany Email: volker.tresp@siemens.com
THE UNIVERSITY OF SHEFFIELD		Hamish Cunningham, THE UNIVERSITY OF SHEFFIELD Sheffield, UK Email: h.cunningham@dcs.shef.ac.uk
VRIJE UNIVERSITEIT AMSTERDAM		Frank van Harmelen, VRIJE UNIVERSITEIT AMSTERDAM Amsterdam, Netherlands Email: Frank.van.Harmelen@cs.vu.nl
THE INTERNATIONAL WIC INSTITUTE, BEIJING UNIVERSITY OF TECHNOLOGY		Ning Zhong, THE INTERNATIONAL WIC INSTITUTE Mabeshi, Japan Email: zhong@maebashi-it.ac.jp
INTERNATIONAL AGENCY FOR RESEARCH ON CANCER		Paul Brennan, INTERNATIONAL AGENCY FOR RESEARCH ON CANCER Lyon, France Email: brennan@iarc.fr
INFORMATION RETRIEVAL FACILITY		John Tait, INFORMATION RETRIEVAL FACILITY Vienna, Austria Email: john.tait@ir-facility.org

Table of Contents

List of figures 7

List of Acronyms 8

1 Introduction 9

2 Rule-Based Reasoning and Web-scale Inference 10
- 2.1 Rule-based Reasoning on the Web 10
  - 2.1.1 Rule-based Inference for Tractable Ontology Languages 11
  - 2.1.2 Rule Languages 12
- 2.2 Rule-based Reasoning in LarKC 15

3 Strategies and Design for Parallel and Distributed Rule-based Inference 19
- 3.1 Preliminaries 20
- 3.2 Parallelization Strategies 21
- 3.3 Data Distribution 22
  - 3.3.1 Map-Reduce 23

4 A Parallel and Distributed Rule Reasoner 25
- 4.1 Parallel Joins by Map-Reduce 25
- 4.2 Prototype Architecture 28
  - 4.2.1 Specific Extensions for Parallel Rule Evaluation 31

5 Conclusion and Future Work 34

References 34

List of Figures

2.1 RIF Dialects 13
2.2 Linked Data instance alignment rules 17

3.1 Basic Map-Reduce Schema 23

4.1 IRIS' Evaluation Pipeline. Components with dashed lines are concerned with parallel rule evaluation itself. 29
4.2 Rule elements in compiled rules. 30

List of Acronyms

OWL	Web Ontology Language
RDBMS	Relational DBMS
RDF	Resource Description Framework
RDFS	RDF Schema
W3C	World Wide Web Consortium
DL	Description Logic
LP	Logic Programming
IRIS	Integrated Rule Inference System
RIF	Rule Interchange Format
SWRL	Semantic Web Rule Language
SKOS	Simple Knowledge Organization System
BioPax	Biological Pathway Exchange
OBO	Open Biomedical Ontologies
ETL	Extract, Transform, Load

1. Introduction

Rule-based reasoning and techniques are relevant for large scale Web inference because they facilitate tractable inference, cover multiple tractable fragments of standard formalisms, and form a convenient modeling approach to many domain specific problem. Applying rule-based reasoning has the advantage that it is possible to reuse and to apply research results, i.e. from the deductive database area, from Logic Programming, and relational database systems in order to meet the scalability requirements posed by Web-scale inference over billions of triples. Therefor, rules have a distinct place in the Semantic Web language stack and their relevance is also reflected by prominent standardization efforts within the RIF working group[1], whose work also draws heavily and are directly grounded in existing research on Logic Programming.

Parallelization efforts for rule-based implementations of RDFS and more recently also tractable, rule expressible fragments of OWL 2 have demonstrated remarkable scalability by using Map-Reduce as underlying programmatic model. However, also research results stemming from parallelization efforts in the database and Logic Programming communities have nearly direct applicability and promising benefits.

In this document we examine the role of rule-based reasoning on the Web and therefore also for LarKC. We examine the relevance of rule-engines in relation to the current work within LarKC as well as (upcoming) standards. In particular we highlight i) requirements for rule-based reasoning in the LarKC use-cases (e.g. for instance alignment in data integration and consistency checking rules that allow to express constraints), ii) rule-based reasoning for subsets of OWL 2, and iii) rule-based reasoning for the rule interchange format (RIF). Following, we identify basic parallelization and distribution strategies. Based on this, we describe an extension to a classical rule engine (IRIS) that allows the parallel computation of joins during rule evaluation, and thus makes it possible to work with huge datasets. Combined with standardization results stemming from the RIF WG this allows to construct a scalable, parallel inference engine for generic RIF-Core rule-sets, which indirectly also facilitates parallel OWL 2 RL and RDFS reasoning. Such a component will effectively add the possibility to use RIF rules along other traditional reasoning components and is inspired by recent work on the parallel and distributed evaluation of Datalog [31] that employs heavily optimized joins in a Map-Reduce architecture. An corresponding implementation is provided as a follow-up to this document in the form of the future deliverable D4.4.2.

The remainder of this document is structured as follows: In section 2 we lay out the background for our work, both in terms of rule-expressible Web standards or fragments of them, such as RDFS or OWL 2 profiles, as well as in terms of the more generic setting of a dedicated rule language, i.e. RIF. Following this overview, we identify relevant strategies and approaches for the parallel and distributed evaluation of rules in Section 3. Based on this we describe the basic approach we adopt in order to parallelize joins in Section 4.1 and then formulate the initial design for our parallel and distributed RIF reasoner, presented in Section 4.2.

[1] http://www.w3.org/2005/rules/wiki/RIF_Working_Group

2. Rule-Based Reasoning and Web-scale Inference

In this section we first outline the focus of our work by providing an overview of rule-based reasoning in general and its importance and role in relation to Web-scale inference. We emphasize this with an exemplary overview of the application of rule-based reasoning within one of LarKC's use-cases (WP7a – Early clinical development). In this sense we cover rule-based reasoning at a high level, how research activities are reflected in current and upcoming Web standards, and we also reflect on the underlying motivation of using rules along other approaches, i.e. along Description Logic knowledge bases.

2.1 Rule-based Reasoning on the Web

Graph-based formalisms RDF(S) were the first standardized ontology languages to represent an ontology's schema and instances. Then, standardized ontology languages based on Description Logic [2], i.e., OWL-DL (later OWL 2) extended RDFS with further expressivity. However, Logic Programming and rules were also introduced because of the limited expressive power of a DL-based ontology language in some situations, such as property chaining, the manipulation of events, states, and actions. Rules on the Web, and thus also rule based reasoning, are central points in the Semantic Web architecture stack, sitting just beside ontologies. Rules can be regarded as complementary to ontologies, which describe multiple objects in a machine understandable manner. Rules in this sense can be seen as means to steer the inference process and ways to combine existing information in order to derive new knowledge.

As mentioned it is possible to employ various kinds of rules in different scenarios (production rules, derivation rules, constraints, etc.), however in this document we mainly focus on derivation rules as those have the most direct applicability within LarKC. Based on this we can concretely identify two main applications for rule-based reasoning and rule engines.

Rule engines in general are relevant because it is possible to employ them to reason with several tractable fragments of standard formalisms, i.e., RDFS [9] or OWL 2 RL [20]. This allows to reuse research results stemming from the deductive database area for reasoning tasks in a Web environment, which usually results in scalable solutions that deal well with large instance sets, e.g., by employing forward-chaining and materialization. Practically this often means that the intended semantics of the language under consideration are encoded as a static set of entailment rules that operate directly on the underlying RDF representation.

However, rule engines cannot only be used to reason with static rule-sets that capture the intended semantics of tractable ontology language fragments, but they can also be used with generic stand-alone rules that are employed alongside with other formalisms. This can lead to hybrid approaches with various degrees of integration of rules and for example Description Logic knowledge bases.

So, in summary we can distinguish two basic types of rule-based reasoning on the Web:

1. Standard inference, e.g. RDFS, OWL reasoning

2. General rule-based inference

However, it is important to point out that there is no conceptual difference between the two, i.e. the second case also covers the first one. From a practical point of view it needs to be noted that the inference rules e.g. for RDFS or OWL 2 RL are fixed. Thus specific optimizations can be applied that are not applicable in a generic rule engine and reasoners may even implement inference without an actual rule engine. For general rule based inference a custom language to express rules and a generic rule engine are required.

Following we briefly examine both of those directions of either applying rule engines for reasoning with tractable fragments of current standards and also standardization efforts toward dedicated rule languages.

2.1.1 Rule-based Inference for Tractable Ontology Languages

Several standardized ontology languages can be effectively dealt with by means of common rule-engines. DLP (Description Logic Programs) [15] are a particular well-known example which can be roughly described as the logical fragment formed by the intersection of Description Logics and Logic Programming. The practical use of DLP is actually twofold: First of all it is a tractable formalism that captures many of the ontologies found on the Web, and secondly it can serve as a basic interoperability layer between Description Logic and Logic Programming based formalisms. WSML [11] is another example and actually a stack of logical languages (variants) that are founded on different formalism. More particularly, the WSML-Flight and WSML-Rule variants are directly based on Logic Programming with various features and different degrees of expressivity. However, it is noteworthy that they both can be mapped to corresponding RIF expressions in a straightforward way.

The recommended minimal representation formalism for LarKC, as outlined in D1.1.4, is OWL 2 RL, which is also amendable to rule-based processing (see below). However, since LarKC is an open and extensible platform several other rule-based formalisms can in theory be considered as well. A requirement for this is only a suitable RDF serialization along specialized plug-ins or a mapping of the particular formalism to a RIF dialect and a standard RIF reasoner – this highlights the importance of RIF since it allows to leverage one common reasoning engine for different purposes. For this reason we will focus mainly on OWL 2 RL in the remainder of this section and consecutively on RIF.

OWL 2 RL OWL 2 [16] extends OWL with several new features which include extra syntactic sugar, property and qualified cardinality constructors, extended datatype support, puning as a simple form of meta-modelling, and furthermore different handling of annotations (they have no semantic meaning anymore).

OWL 2 DL is an extensions of OWL-DL and corresponds to the Description Logic $\mathcal{SROIQ}$. However, OWL 2 is not given a semantics by mapping to $\mathcal{SROIQ}$ (since puning not being available in it) but rather directly on the constructs of the functional-style syntax for OWL 2. Similarly to OWL, OWL 2 again comes in several different flavors : OWL 2 Full (which is an extension of the RDFS semantics again), OWL 2 DL, OWL 2 EL, OWL 2 QL, and OWL 2 RL. The last three are so called *profiles*(see [20]), which are fragments of OWL 2 that eliminate specific language elements (expressivity) for more efficient reasoning and in turn scalability. Most of the profiles are defined by

enforcing syntactic restrictions and thus also limiting the expressivity of OWL 2 – in that sense they all trade expressive power for efficient reasoning.

From the perspective of rule-based reasoning especially OWL 2 RL is interesting. It is a fragment which is customized to support reasoning with rule engines by only considering objects that are explicitly named in the knowledge base. It is a profile that is intended to form a proper extensions of RDFS while still being computationally tractable. As such it realizes a weakened form of the OWL 2 Full semantics and is very similar in spirit to DLP [15] and pD* entailment [30]. OWL 2 RL is a syntactic OWL 2 subset whose semantics are defined via a partial axiomatization of the OWL 2 RDF-based semantics [26]. Practically this means that its semantics are defined by a number of first-order implications (rules) which can be used as a direct starting point for implementations. A typical example of such a rule is:

```
IF      ?p rdfs:subClassOf ?t
        ?t rdfs:subClassOf ?c
THEN    ?p rdfs:subClassOf ?c
```

In turn, the OWL 2 RL profile is directly amendable (by design) to an implementation on common rule engines or extended RDBMS. Common reasoning tasks, such as consistency checking, subsumption checking, instance checking, or query answering, can be solved in polynomial time with respect to the size of the data.

For practical purposes OWL 2 RL's entailment rules have also been defined as a fixed set of RIF Core rules [10]. As an alternative to the direct application of those rules, which can often be considered to be fairly inefficient, is it is also possible to map any OWL 2 RL ontology into a particular, custom set of RIF Core rules (again see s [10]). This translation process is based only on the TBox axioms of the respective OWL ontology and the resulting rule set can be applied to any knowledge-base that differs only through its ABox assertions – this translation of the ontology schema to a custom rule set does not need to be repeated as long as only instance data changes. However, the important aspect is that, whatever technique is applied, any rule engine able to consume RIF Core rules can be employed for OWL 2 RL reasoning as well. Thus, a scalable RIF inference engine also has the immediate benefit of being able to reason over OWL ontologies with a practical degree of expressivity and can at the same time deal with comparatively large instance sets as well.

2.1.2 Rule Languages

SWRL In terms of Web standards, SWRL [13] is roughly based on combining OWL-DL with Datalog [31]. Thus Horn-like rules (although based on First-Order Semantics) can be used in conjunction with existing OWL knowledge bases. This approach maintains the expressivity of OWL DL based on an extension of the model-theoretic semantics of OWL (and SWRL also is based on the open world assumption). In fact it could be said that every SWRL rule is an OWL axiom, albeit in a more generalized and less restricted form (thus SWRL also supports monotonic inference only). This expressivity however leads to an undecidable (in fact semi-decidable) formalism for key inference tasks and thus practical implementations rarely support the full specification. Decidability can be regained by placing restrictions on the SWRL rules, resulting in DL-safe rules [21]. These restrictions mandate that each variable in rules can only

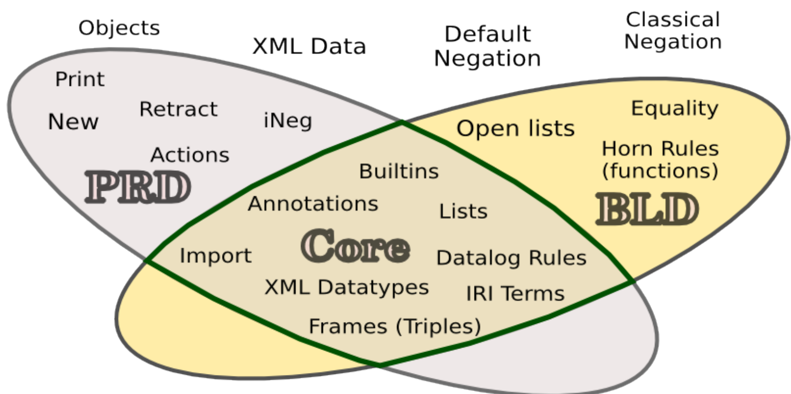

Figure 2.1: RIF Dialects

bind to explicitly named individuals or more precisely that variables that occur in a rule also need to occur in a non-DL-atom in the rule body.

RIF – Rule Interchange Format In addition to the Semantic Web Rule Language (SWRL), the Rule Interchange Format (RIF) [8] is a Web standard that defines an interchange language for rules among different rule systems. As such it builds on and develops the existing range of specifications such as RDF/RDFS and OWL. In contrast to SWRL, which was designed as a rule language as an extension to OWL, RIF is much broader in scope and thus it would e.g. be possible to exchange rules between systems that implement SWRL.

RIF focuses on the definition of various *dialects* (see Figure 2.1 for an overview of features[1]) within a common framework in order to easy the exchange of rules rather than trying a single rule language because of the different paradigms for using rules in knowledge representation and business modeling. Defined within a common framework, those dialects share as much of the existing syntactic and semantic building blocks as possible and thus automatically define a minimal level of interoperability. Furthermore a common framework eases the definition of new RIF dialects as a syntactic extension of an existing standard. Present and future RIF dialects are expected to share datatypes (e.g. integers, strings, booleans), built-in functions (e.g. arithmetics, string manipulation), and built-in predicates as defined by the RIF Datatypes and Built-Ins (RIF-DTB) specification. Following we give a brief overview of the different predefined dialects and their respective purpose.

RIF-FLD (Framework for Logic Dialects): RIF-FLD is not really a dialect on its own but the common framework used to define specific dialects through specialization. Its main purpose is to lower the effort required to define new dialects within its common conceptual framework.

RIF-BLD (Basic Logic Dialect): RIF-BLD is a specialization of RIF-FLD capable of representing definite Horn rules with equality and including function symbols, enhanced with a number of syntactic extensions to support expressive features

[1] Taken from `http://www.w3.org/2009/Talks/1105-ruleml`

such as objects and frames, internationalized resource identifiers (IRIs) as identifiers for concepts, and XML Schema data types.

RIF-PRD (Production Rule Dialect): RIF-PRD is intended to serve as exchange format for various production rule systems. Since the meaning of production rules is often defined in an ad-hoc fashion and not based on logic, RIF-PRD also is not part of the logical RIF dialects and stands apart from them.

RIF-Core (Core Dialect): RIF-Core is the simplest variant of RIF. Formed as a common subset of both RIF-BLD and RIF-PRD , thus enabling limited rule exchange between logic rule dialects and production rules. This results in a simple rule language that allows to define Horn rules without function symbols, basically corresponding to Datalog, including common safety features built in so that the rules can be executed by a forward chaining engine. Despite its simplicity RIF-Core is expressive enough to capture many practical use-cases, including the semantics of OWL 2 RL.

A typical scenario, and one that we deliberately aim at, is the use of RIF along with RDF/OWL. Practically this means that arbitrary rules (corresponding to a specific RIF dialect) can be used with RDF data and/or RDFS or OWL ontologies. The relevance of such a scenario is also documented in a specific LarKC use-case.

A RIF document that refers to (imports) RDF graphs and/or RDFS/OWL ontologies, or any use of a RIF document with RDF graphs, is viewed as a combination of a document and a number of graphs and ontologies. RIF provides a mechanism for referring to (importing) RDF graphs and a means for specifying the context which corresponds to the intended entailment regime in RDF/RDFS. A RIF-OWL-combination consists of a RIF document and a set of RDF graphs, analogous to a RIF-RDF combination, because the syntax for exchanging OWL ontologies is based on RDF graphs. Due to the relevance of the combination of RIF rules and RDF data a specific document within the RIF standardization stack specifies this interaction [24]. On the other hand, it is also possible to embed RIF rules in RDF data as literal due to RIF's XML serialization.

Practically there is a very intuitive correspondence between statements in RDF graphs and certain kinds of formulas in RIF. Namely, there is a correspondence between RDF triples of the form $\langle s, p, o\rangle$ and RIF frame formulas of the form $s'[p' \rightarrow o']$, where s', p', and o' are RIF symbols corresponding to the RDF symbols s, p, and o, respectively. This means that whenever a triple $\langle s, p, o\rangle$ is satisfied, the corresponding RIF frame formula $s'[p' \rightarrow o']$ is satisfied, and vice versa. The following example illustrates the interaction between RDF and RIF:

The following RDF statements

```
ex:john ex:brotherOf ex:jack.
ex:jack ex:parentOf ex:mary.
```

state that ex:john is a brother of ex:jack and ex:jack is a parent of ex:mary.

A RIF rule

```
Forall ?x ?y ?z (?x[ex:uncleOf -> ?z] :-
     And(?x[ex:brotherOf -> ?y] ?y[ex:parentOf -> ?z]))
```

says that whenever some x is a brother of some y and y is a parent of some z, then x is an uncle of z. From this combination the RIF frame formula :john[:uncleOf -> :mary], as well as the RDF triple :john :uncleOf :mary, can be derived.

Furthermore, and as pointed out, there is a very clear relationship between the OWL 2 RL profile and RIF-Core, which is facilitated by RIF's compatibility with RDF data: OWL 2 RL is defined as a partial axiomatization of the OWL 2 RDF-based semantics is given in the OWL Profiles document as a set entailment rules. These rules can be translated into RIF-Core rules either as a static rule set or on a per-ontology basis as discussed [10]. So, obviously, the broader focus of RIF, that allows to specify arbitrary rules, also covers cases where rule engines are used to reason with tractable ontology languages or respectively tractable fragments – as long as a suitable set of entailment rules can be specified.

This also gives users the flexibility to discard specific OWL 2 RL language elements which they do not care about by simply dropping the corresponding rules. Thus it is trivially possible to trade reasoning complexity for performance by tuning the rule set to be used. The same approach obviously also works with RDFS as a starting point (and a reduced RIF rule set), which in turn makes it possible to gradually increase or decrease the required reasoning complexity/performance along a wide spectrum.

This leads to the conclusion that (at least) RIF-Core through a scalable rule-based implementation would be a very desirable addition to the LarKC platform, both from a theoretical as well as from a practical point of view, as it immediately allows i) "standard reasoning" along this spectrum (RDFS up to OWL 2 RL), and ii) the combination of custom (and potentially user-defined) rules along with this standard reasoning. Following we illustrate this potential applications of rule-based inference in a concrete LarKC use-case.

2.2 Rule-based Reasoning in LarKC

While the previous section examined rule-based reasoning and its applicability in the context of current Web standards in general, this section focuses on a concrete LarKC use-case concerned with early clinical development. This use-case performs semantic data integration for the biomedical domain resulting in LinkedLifeData[2]. For this purpose it uses LarKC as a platform to integrate and interpret data from various heterogeneous sources (UniProt, PubMed, EntrezGene, etc.) and produces and aggregated dataset resulting in over 4 billion RDF statements. Reasoning is used for various purposes within this process and in turn we will analyze the applicability and potential benefits of rule-based reasoning for these various facets of the use-case which require inference.

Schema inference The integrated data sources encode special semantics, for example the BioPAX domain ontology uses OWL-DL[3] for this purpose. Data sources distributed in OBO format[4] can be encoded using SKOS [19]. In summary, the minimum level of schema inference required within this specific use-case roughly corresponds to a combination of RDFS, SKOS, and OWL. However, only few useful language elements

[2] http://linkedlifedata.com/
[3] http://www.biopax.org/
[4] http://www.geneontology.org/GO.format.obo-1_2.shtml

from OWL-DL are actually. Thus it is feasible to employ a rule engine to handle the basic schema inference and for example disjoint constructs could be substituted with simple consistency checking rules. In turn only lightweight inference that is amendable to processing on a rule engine is required, for example typical reasoning tasks according to the SKOS schema specification are:

```
<A> skos:broader <B> .
<B> skos:broader <C> .

entails

<A> skos:broaderTransitive <B> .
<B> skos:broaderTransitive <C> .
<A> skos:broaderTransitive <C> .
```

Another example of schema inference, used for the purpose of semantic data integration, which can be handled by simple rule-based reasoning is the alignment of different biomedical thesaurus:

```
<A> skos:broadMatch <B> .

entails

<A> skos:mappingRelation <B> .
<A> skos:broader <B> .
<A> skos:broaderTransitive <B> .
<A> skos:semanticRelation <B> .
<A> rdf:type skos:Concept .
<B> rdf:type skos:Concept .
```

Transformation reasoning / ETL / Mapping rules: Extraction, Transformation, Loading (ETL) is a typical phase in the generation of data warehouses. RDF warehousing requires similar operation to address the variety of different data modeling approaches. Based on more than 20 different RDF database representations several integration patterns required to interconnect related resources can be identified in our specific use-case. The different patterns applied in the LinkedLifeData alignment process are depicted in Figure 2.2 and can be captured in *mapping rules* that allow to link i) equal instances, ii) resources with a specific form of semantic relationships. . The solid lines and the coressponding text of the captions (used either as part of the URI or literals) designate the criteria for linking the information. The specified mapping rules are not universally applicable for all RDF types and they are applied only to subsets of the information. The process of the subset selection and the rule application is manually controlled.

As of now, no single efficient solution to implement all the mapping rules through a single component is available, therefore their processing is divided as follows (see D7a.3.1 for more details):

1. Namespace mappings in a particular dataset are all covered by the URI generation convention

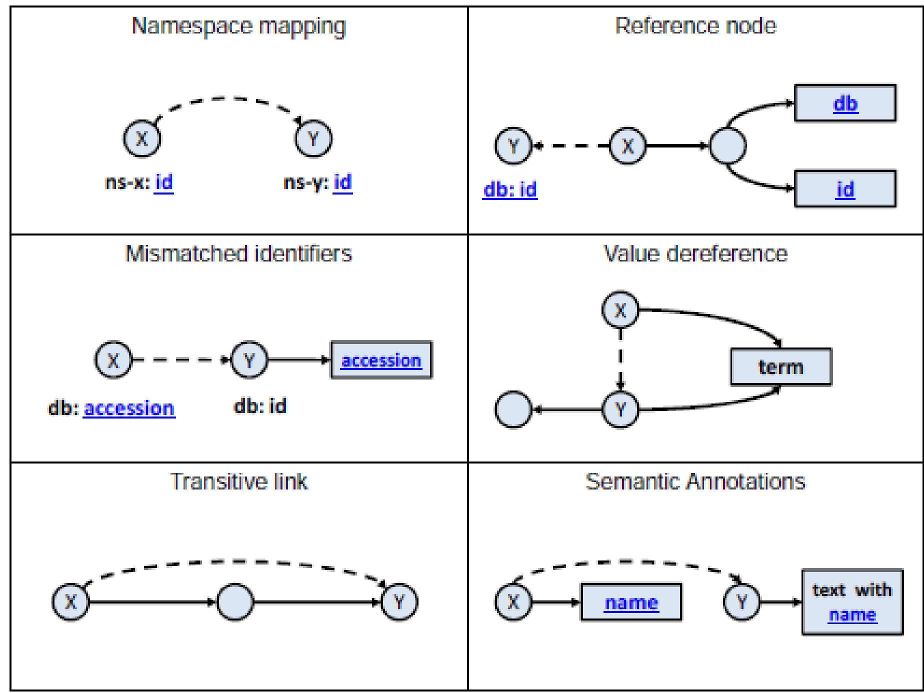

Figure 2.2: Linked Data instance alignment rules

2. Reference node, Mismatched identifiers, and Value dereference are processed by a custom implementation Java reasoner
3. Transitive links are declared as custom OWLIM inference rules and OWL transitive properties
4. Semantic annotations are processed by a GATE pipeline and executed in parallel.

The final goal for all this patterns (covered through specific rules) is to generate explicit links between two instances X and Y based on existing information in the model. After the necessary transformation are asserted, SPARQL may be used to query the integrated data model. As of now no generic component is handled to cover all those mapping rules but rather a combination of specific SPARQL queries (which is not able to express mapping rules fully due to the lack of proper string concatenation and substring functions) and custom crafted code is necessary.

Consistency checking rules The semantic integration process aggregates information from multiple and potentially inconsistent data sources. The management of this complex information model needs 1) control mechanisms to guarantee certain level of consistencies and soundness of the data and 2) polynomial inference time that guarantees predictive reasoning behavior. The data integration process is characterized by a constant flow of new information in the knowledge base (e.g., new versions, additional data sources and etc). Thus, a special form to control the consistency or the correctness of the knowledge base is required.

For example the SKOS schema semantics defined in [19] specifies a light-weight vocabulary to express thesauri, taxonomies, classification schemes and subject heading

systems. Using OWL 2 RL as ontology language and a rule-based reasoner it is very easy to express that the next two following two statements are in fact inconsistent:

```
<instance> skos:prefLabel "label1" ;
 skos:prefLabel "label2" .
```

The previous rule may be defined as:

```
IF ?skosConcept skos:prefLabel ?label1
   ?skosConcept skos:prefLabel ?label2 (label1 != label2)
CHECK ?skosConcept rdf:type owl:Nothing
```

The correct way to read the previous consistency rules is every concept which has two different label values and is not of type `owl:Nothing` is invalid. For simplicity and better control the `owl:Nothing` type statement is used to denote RDF resources which are known to be inconsistent. As a concrete language to express such a rule for example RIF could be used.

3. Strategies and Design for Parallel and Distributed Rule-based Inference

As outlined in the previous section rule-based reasoning techniques have several potential applications within the modern language stack of the Semantic Web. They can be used with a set of pre-defined static rule sets, as in the case of RDFS or OWL 2 RL, or also in the form of custom user-specified rules (which can be exchanged e.g. via RIF). Within this section we examine various strategies that have been proposed for the parallelization and distribution of inference in order to meet practical scalability requirements.

Recently several solutions have explored parallelization as a solution to the challenges posed by Semantic Web reasoning and especially rule-based reasoning with static rule sets. Working with a static rule set, e.g. for RDFS or OWL 2 RL reasoning, often allows very specific optimizations in order to minimize the overhead causes by data distribution and communication.

The authors of [28] and [29] demonstrate a technique for parallel OWL inference based on rule-based reasoning. One of the key challenges pointed out there is that partitioning rules and/or data is a non-trivial problem in order to minimize the duplication of computation and the amount of data communicated among processors. Therefore, different data- and rule-partitioning schemes that facilitate OWL-Horst [30] (which is defined through entailment rules that are basically negation-free Datalog) reasoning are proposed. The authors address the aforementioned issues by taking advantage of the particular nature of the entailment rules. More particularly, this means that i) dependencies between entailment rules are taken into account (i.e. a *dependency graph* is constructed), and ii) most of the entailment rules are essentially *single-join rules* (rules which have two sub-goals in the body of the clause and both these sub-goals share a variable) – rules involving more than one join in particular are required for universal and existential quantification.

More recently [32] and [33] follow a similar pattern in order to compute RDF and OWL-Horst closures over 100 billions of triples by forward-chaining, although they employ Map-Reduce [12] as a framework to facilitate those massive computations. A noteworthy point is that, again, only rules involving a single join are parallelized and for rules that require multiple joins, some of the joins are performed in memory. Furthermore several other, detailed optimizations are applied, which are customized for the specific rule set at hand. [35] essentially follows a similar approach for the computation of big RDFS closures. Also, [22] applies Map-Reduce as a framework for reasoning with the tractable Description Logic $\mathcal{EL}^{++}$, and more particularly to classify $\mathcal{EL}^{++}$ontologies, by evaluating a specific set of completion rules in parallel. [18] applies Logic Programming based techniques in order to drive a resolution based system for the more complex Description Logic $\mathcal{SHIQ}$, which is again takes the particular structure of the underlying logic into account in order to parallelize instance retrieval.

Several key points can be summarized:

- In the outlined approaches parallelism is often achieved by i) analyzing rule dependencies, and ii) by custom tailoring evaluation algorithms for specific sets of rules.

- Parallelization schemes have to minimize communication cost and the amount of redundant computation in order to be practically feasible, i.e. special distribution and partitioning schemes need to be employed.
- Many of the rule-sets that are practically parallelized are rather inexpressive. They can usually be captured by simple Datalog/RIF-Core and only some rules require multi-way joins, although those are usually considered difficult to implement efficiently.
- Map-Reduce, and thus essentially a centralized, shared-nothing approach, is used in several cases as an efficient and simple programmatic framework. In several cases multiple rounds of Map-Reduce and rule applications need to be performed, until a fix point is reached.

Several of those points carry over to a generic rule system, i.e. one that allows arbitrary RIF-Core/Datalog rules. For example the relevance of efficient and parallel join computations as well as the requirement to minimize data transmission, and in particular also the benefits of the general Map-Reduce paradigm for large scale data processing. However, at the same time it is not easily possible to apply optimizations that target a specific rule set for a general purpose reasoner that accepts generic RIF rules as input – especially since correct and efficient data partitioning for multi-way joins becomes more complex.

After this overview of parallelization strategies aimed at applying rule-engines for "standard reasoning" we now proceed to introduce the strategies for the parallel evaluation of generic rules by first introducing some preliminaries and then proceed to identify useful basic approaches.

3.1 Preliminaries

As a starting point for a more comprehensive rule-based reasoner that evaluates rules in a distributed and parallel fashion, we explicitly focus on rather simple Datalog [31] programs – however of sufficient expressivty to cover RIF Core [7] and thus indirectly also OWL 2 RL (see previous sections). A Datalog program is a finite set of Horn-clause rules of the form

$$H_0 : -B_1, \ldots, B_n$$

in which B_i is a positive literal or predicate of the form $P(x_1, \ldots, x_m)$. In such a Datalog rule H_0 is called the *head* of the rule and $B_1, \ldots, B_n$ the *body* of the rule. It is possible and often convenient to divide the predicate symbols in a program into two (disjoint) sets:

1. Extensional (or base) predicates
2. Intensional (or derived) predicates

Base predicates cannot appear in the head of any rule in a program. The Extensional Database (EDB), the set of extensional predicates, can also be called *facts* and form the raw input data for a program evaluation. The set of intensional predicates is also called the the Intensional Database (IDB). Furthermore, predicates in the body of a rule are also called *subgoals*.

A simple approach to evaluate a Datalog program in a bottom-up fashion simply involves evaluating the respective rules for as long as new tuples can be derived. This implies finding a substitution for variables by constants (grounding them) such that, if the body of the rule is satisfied, new values are added to the database according to the head of the rule – the particular instantiation is *applicable*. Practically this usually implies to a series of joins involving the relations backing the individual subgoals in the body of the rule. Grounding variables in a rule yields a particular instantiation of the rule. For example consider the following rule

$$Q(x, y) : -P(x, z), B(z, y)$$

and assume that a, b, c are values for tuples in the database, then

$$Q(a, c) : -P(a, b), B(b, c)$$

is an instantiation of the initial rule. If $P(a, b)$ and $B(b, c)$ are both true with respect to the data contained in the knowledge base then the instantiation is applicable and $Q(a, c)$ can be derived.

It is crucial to realize that joins are a fundamental operation for that purpose, and that given a large amounts of facts in the EDB, rule evaluation can be time consuming largely due to the time taken up for those operations unless deliberate optimizations are applied.

This is even more so in a Web setting, where inference can be assumed to be primarily data driven. For this reasons several strategies have been developed to address the challenge of rule-based reasoning over large fact bases through parallelizing the inference process

3.2 Parallelization Strategies

As pointed out in the previous section we can essentially distinguish different types of predicates for practical purposes and thus separate them in an intensional and extensional part. i.e. *facts* and predicates for which new tuples will be derived. In a similar, but slightly different fashion it is also possible to distinguish two basic approaches as starting point for a parallelization strategy, namely:

1. Rule/Program Partitioning

2. Data Partitioning

Rule/Program Partitioning: Basic approaches aiming at the parallel execution of a program through partitioning it by means of syntactic characterizations into independent parts, which can be evaluated independently. Such programs can be called *decomposable* and can be evaluated in a distributed and parallel fashion without any communication between compute nodes (a “shared-nothing architecture”). Such decomposition criterion and algorithms have been well studied (e.g. see [36], or [27]) and enable the parallel computation of independent parts of the rule-base without requiring redundant data transmission and re-computation of tuples.

However, such techniques come with some inherent drawbacks attached. First of all, only restricted Datalog programs can be decomposed and thus evaluated in

parallel. Furthermore, the property of being decomposable is undecidable for general Datalog programs. Finally, the number of nodes that can be used in the parallel evaluation of a program intrinsically depend on the number of parts into which it can be decomposed. So, while such approaches offer a simple solution and advantages, they are only applicable for a limited amount of cases, and thus more “fine-grained” methods have been proposed.

One such possibility is *rule decomposition*. Rules of a program are typically evaluated in a top-down style (backward-chaining) and therefore focus on only deriving information that is relevant to a particular query and only working on “relevant facts” (see [17], [4]). Typically, such strategies decompose the rules of a program into concurrent modules that are assigned to different nodes. It is important to note, that such approaches do not only decompose programs but individual rules, i.e. different parts of a rule will be assigned to separate nodes – basically a node evaluates one or a conjunction of subgoals of a rule. However, rule decomposition suffers similar drawbacks to program decomposition. Namely the amount to which a set of rules can be parallelized again depends on the specific structure of the rule(s), and in turn it is not possible to scale rule evaluation to an arbitrary number of available computation nodes. Furthermore, since the structure of a rule determines how it can be distributed to nodes, it is not easy to achieve a balanced work distribution and apply further optimizations.

Data Partitioning: Another fundamental strategy relies on partitioning data and in turn also on the *partitioning of rule instantiations*, with examples being [14] and [38]. Such an approach basically results in a bottom-up evaluation that parallelizes the evaluation of a programming by assigning subsets of the data to different nodes. In turn each node executes the complete program but with less data, and therefore less rule instantiations.

[36] introduces the notion of a discriminating predicate, which is appended to each rule and used in combination with a hash function in order to assign each rule instantiation to a specific node. Data partitioning can also be applied directly at the level of the relation algebra operators, as for example in [6] or [23]. Such strategies work well for relatively simple rules/programs since it can be applied in a very modular way without much concern about the overall structure of the underlying program, however they have the potential to cause redundant data transmission without further optimizations in place. More recent work, such as [1] or [37] aim at the direct parallelization of relational algebra operators, and apply Map-Reduce as framework for this purpose.

3.3 Data Distribution

As pointed out, communication overhead and balanced distribution of data plays a critical role in order to obtain acceptable parallel performance. Workload should be fairly divided among nodes and duplicate computations within the nodes should be avoided. On the other hand, data transmission becomes a fundamental issue for the overall performance of the rule evaluation. This basic trade-off between redundant computation at individual nodes and the required communication is in-depth discussed in [14]. The main goals of a suitable data distribution can be summarized as following:

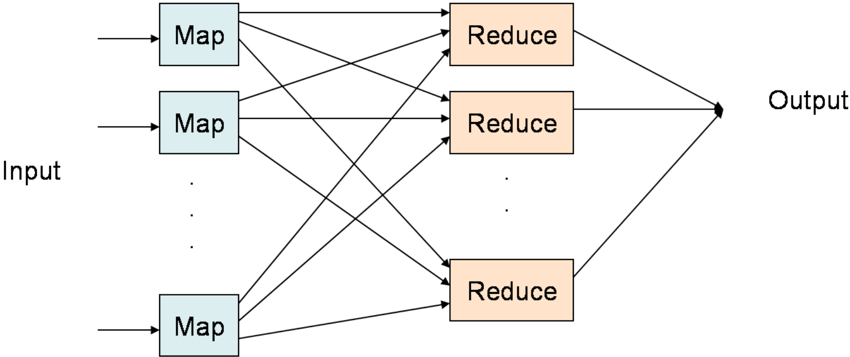

Figure 3.1: Basic Map-Reduce Schema

- Minimization on the number of transmissions and the total amount of data transmitted for a complete evaluation of a set of rules.
- Minimization of redundant communication and computation.

These points also form the underlying motivation for some of the optimizations discussed before; to devise a specific order for rule evaluation based on inter-dependencies between rules, and to reduce the number of Map-Reduce iterations required until a fix-point is reached. For generic rule sets, a cost measure can be devised to optimize according to the aforementioned criteria. As a concrete solution to meet this goals we have chosen to adopt a centralized, shared-nothing approach oriented toward a Map-Reduce like model and thus will briefly explain the underlying concepts.

3.3.1 Map-Reduce

Map-Reduce [12] is a programming model for distributed processing of data on clusters of machines (each machine being called a node). Map-Reduce processes data in two phases: the map and the reduce phase, with both phases being inherently parallel. Generally, the underlying idea is that the data to be processed is divided into multiple chunks, and each chunk is assigned to an idle node for processing

There are two different types of nodes, and each type has its own function and corresponds to one of the two processing phases: map nodes and reduce nodes. Each mapper and reducer implements specific user-defined functions, namely the map and reduce functions. The general processing schema is depicted in Figure 3.1.

Map phase: During the map phase, each mapper reads input, or potentially receives it from a dedicated *master node*. Map nodes then generate intermediate output according to a user-defined *map function*. The map function generally accepts a key-value pair and returns a set of key-value pairs (the intermediate output) according to the concrete map function. The functionality of Map nodes can be represented as

$$\text{Map: } (k_1, v_1) \rightarrow \text{list}(k_2, v_2)$$

The map functions output pairs are further propagated to subsequent phases where they are partitioned/grouped and sorted by their keys. Grouping is achieved using a partitioner which partitions the key space by applying a hash

function over the keys. The total number of partitions is the same as the number of reduce tasks (reducers).

Reduce phase: Reduce nodes further consume intermediate output. They group values by key, and more specifically, each reducer receives a corresponding partition in a fashion that all values with the same key are passed to the same reducer. The fetched map output pairs are merged based on the same key. The newly structured pairs of key/list are propagated to the user defined reduce function, which process them. The general process can be represented as

$$\text{Reduce: } (k_2, \text{list}(v_2)) \rightarrow \text{list}(v_3)$$

There are several prominent Map-Reduce implementations readily available, e.g. Hadoop[1], and developers usually only need to define the map and reduce functions without being concerned about lower level tasks.

[1] http://hadoop.apache.org/mapreduce/

4. A Parallel and Distributed Rule Reasoner

In the previous sections we analyzed related work, relevant standardization approaches, and parallelization and distribution strategies. In particular, Section 3.1 showed the fundamental relevance of joins operations for rule evaluation, especially when considering a large number of facts, and Section 3 explained how various approaches employ Map-Reduce in order to parallelize the joins required for a static rule-set (e.g. for RDFS).

We now proceed to describe the particular parallelization scheme that we adopt for the implementation in a prototype and subsequently as a component for the LarKC platform (plug-in). For this, we outline the theoretical foundations in the following section. This parallelization and optimization approach deals with *arbitrary* joins for generic rules (based on results in [1]), and also leverages Map-Reduce as data distribution scheme.

This technique will be coupled with the existing IRIS[1] [5] rule engine in order to facilitate the parallel and distributed evaluation of individual rules, coupled with more traditional and readily available optimizations. Thus, in Section 4.2 we subsequently explain the IRIS architecture and how parallel rule evaluation can be smoothly integrated. Extending IRIS with parallel rule evaluation in a Map-Reduce fashion essentially amounts to a centralized parallel shared-nothing architecture, where one reasoning core performs optimizations orthogonal to parallelization, parallelizes joins required for individual rules, and is in charge of the communication with compute nodes.

Our parallelization strategy amounts to data partitioning and facilitates the bottom-up evaluation of Datalog programs (and thus RIF-Core). This is a suitable approach for the particular scenario that we envision, i.e. massively data driven inference using only comparatively simple rule-sets, which would massively limit the applicability of rule and program decomposition techniques, and for which the complexity of potential hybrid solution likely out-weights the benefits. Moreover, a direct parallelization of joins seems favorable, since such a technique can easily be applied along with other miscellaneous optimizations (e.g. Magic Sets [3] or other rewriting methods), and can be seamlessly integrated in an existing rule engine.

4.1 Parallel Joins by Map-Reduce

Basic Joins Basic joins needed for rule evaluation can be parallelized using Map-Reduce in a straightforward way (see [37] for variants). For example, considering the following rule

$$P(?x) :- Q(?x, ?y), R(?y, ?z)$$

we can parallelize the natural join required between relations Q and R in the following way:

Map: We first associate each tuple from either relation with a *key* that is the value of its ?y-component. For this purpose a set of available map nodes turn each tuple (a, b) from Q into a *key-value* pair, with *key* $= b$ and *value* $= (a, Q)$. The underlying relation is included in the value, so that in the reduce phase

[1] http://www.iris-reasoner.org/

it is possible match only tuples from R with tuples from Q, and not a pair of tuples from R or a pair of tuples from Q – we only want to combine tuples from different relations. A number of map processes also performs the same operation with every tuple (b, c) from relation R, resulting in a key-value pair with key $key = b$ and $value = (c, R)$. Note that this partitioning can already be parallelized.

Reduce: The reduce phase combines tuples from the two relations that have a common ?y-component. Therefore, tuples with a specific ?y-value have to be sent to the same reduce process. Assuming that k reduce nodes are available, we choose a hash function h that maps ?y-values into k buckets, so each possible hash value corresponds to one of the available reduce nodes. In other words, each map node sends key-value pairs with $Key = b$ to the reduce node associated with $h(b)$. Reduce nodes each process their share of the overall data in parallel by computing a traditional natural join and return the joined tuples (a, b, c), which can then be merged and further processed.

Multi-way Joins As mentioned in previous sections (see Section 2.1 in particular), rules will often require multiple joins in one rule – as a practical example consider the OWL 2 RL rule set (existential and universal quantification) [20]. Datalog rules can often be expected to have rule bodies with several subgoals, as an example consider the following rule:

$$P(?x) : -Q(?w, ?x), R(?x, ?y), S(?y, ?z)$$

Clearly, the evaluation of such a rule could be implemented by a sequence of basic joins, with each basic join requiring a Map-Reduce operation. However, this is cooperatively inefficient. It does not meet the requirements outlined in Section 3 since it causes unnecessary communication overhead. This especially becomes an issue when considering a semi-naive evaluation algorithm, which has to deal with complex expressions with many relations and their increments.

Therefore it is desirable to join all three relations in a single Map-Reduce iteration. In a similar way as to the two-way join, each tuple R is only sent to one reduce node, however in order to be able to perform joins, tuples from Q and S need to be replicated to several reduce nodes. This duplication of data increases the communication cost for single Map-Reduce operation required to perform the joins but on the other hand we do not have to communicate intermediate results. It is crucial to optimize this duplication in order to minimize this overhead. [1] outlines an optimization strategy based on a cost measure for the required duplication and we subsequently adopt this approach as a practical solution.

The general approach to evaluate a multi-way join works similar to a two-way join as follows (considering the rule above): Assume k reduce processes. An arbitrary number of map nodes again partition the input data and send tuples of Q and S to many different reduce nodes, while tuples from R are *only* sent to nodes where they are required to perform joins.

Again we chose a hash function h with a range of $1, \ldots, \sqrt{k}$. Reduce processes are associated with a vector of hash values now, e.g. (i, j) where i and j are between $1, \ldots, \sqrt{k}$. The dimensions of the vector depend on the number of joins required for a particular rule.

Each tuple $R(b, c)$ is only sent to the one reduce process $(h(b), h(c))$. Tuples $Q(a, b)$ are sent in a duplicate fashion to all reduce nodes $(h(b), x)$ for any x since we need to ensure that the right tuples to actually perform the join are available at a node. In the same way, tuples $S(c, d)$ is sent to all reduce nodes $(y, h(c))$ in order to facilitate the second join for the rule.

Each reduce process again simply computes the natural join of the tuples that it receives and returns its results, which can be merged. This is possible because for any three concrete tuples $Q(a, b)$, $R(b, c)$, and $S(c, d)$, they will automatically be routed to the correct reduce process $(h(b), h(c)$.

Optimizing Data Distribution: This basic approach to parallelize join operations partitions data evenly according to the total number of available reduce processes, i.e. the same number of buckets ($\sqrt{k}$) was used for each join variable since the hash function h has a range of $1, \ldots, \sqrt{k}$ for each of them. This basically amounts to reserving a certain share of the range of a hash function for each variable. Denoting these shares as a and b for two variables (considering the above rule again), the product of the shares always has to amount to the total number of reduce processes: $a * b = k$

This can serve to construct a basic cost measure for a specific join that captures the amount of redundant communication caused by tuples sent from map to reduce processes. The total number of tuples in such a communication can be expressed as

$$s * a + t * b$$

where a and b are the shares assigned to variables, and s and t are the sizes of the relations Q and S respectively. This expression can be minimized under the constraint that $a * b = k$ (since we assume a fixed number of reduce nodes). An applicable solution for this minimization problem is the common method of Lagrangean multipliers, which is a general purpose strategy to find minima and maxima of functions that takes constraints into account.

For this we can start with the following expression

$$s * a + t * b - \lambda(a * b - k)$$

take derivatives with respect to the variables, a, b, and set the resulting expressions equal to 0. The result is two equations (for two share variables):

$$s = \lambda b$$
$$t = \lambda a$$

Multiplying each equation with the variable missing on the right side, and under the constraint that $a * b = k$ we arrive at the following equations:

$$s * a = \lambda k$$
$$t * b = \lambda k$$

From here it is possible to solve for λ and subsequently the shares a, b (based on the two equations). Substituting these values into the original cost expression $s * a + t * b$ gives the minimally required communication between map and reduce processes. While a and b do not need to be integers, they generally give a good approximation in order to minimize the communication required to compute a particular join.

A detailed, complete algorithm that generalizes the outlined method, and minimizes the amount of redundant communication caused by parallelizing the joins for an arbitrary rule in the aforementioned manner, is presented in [1] along with potential heuristics.

4.2 Prototype Architecture

In this section we now proceed to illustrate the architecture of the prototype implementation that integrates the approach explained in the previous section.

A prototype implementation of the outlined strategy, that leverage Map-Reduce in order to parallelize joins, is constructed based on the IRIS Datalog reasoner along with a suitable RIF parser that translates imported RIF-Core rules to IRIS' internal rule representation – RIF-Core rules can easily be embedded along with RDF data (e.g. through their XML serialization). Furthermore, since we target RIF-Core, we can simplify some aspects of the existing IRIS reasoner, since we do not have to take negation into account and thus e.g. do not require stratification of rules.

IRIS is a highly modular reasoner and comprised of a number of loosely coupled components that are implemented according to well-defined interfaces. Due to this modular structure we can easily integrate this parallelization approach with the existing bottom-up evaluation strategy and apply orthogonal optimizations. Broadly speaking, an evaluation strategy represents a particular combination of processing elements. Those components interact in a pipeline fashion as distinct steps in order to evaluate a particular rule program. A high level overview of the involved processing components is depicted in Figure 4.1 and also shows that only the last two steps require attention in order to parallelize the rule evaluation – for the initial processing steps existing modules in IRIS can be re-used. The individual elements are briefly explained subsequently.

Program optimizations In the initial phase of the evaluation of a set of rules we apply program wide optimizations steps. A very basic example for such an optimization is rule-filter, which involves analyzing the complete program and building a dependency graph between all rule predicates and removing those rules that can not influence the result for a particular query, thus reducing the size of the minimal model computation. This optimization step is akin to program decomposition techniques and can also be extended to identify independent program parts for further parallelization.

Rule Safety Processing An unsafe rule is one in which a variable is used, but has no binding. In essence, the entire range of possible values must be substituted for this variable, which is clearly impractical. Unsafe rules are therefore particularly problematic for bottom-up evaluation techniques that do precisely this, i.e. substitute known values into variables of rule body predicates (performing rule instantiations). As in traditional Datalog this problem is addressed by appropriate safety restrictions placed on RIF-Core rules [7].

The standard rule-safety processor of IRIS simply examines each rule and indicates if any rule is unsafe and exactly why it is unsafe. A program containing an unsafe rule results in a specific exception being thrown containing a message explaining which rule is unsafe and which variables are problematic.

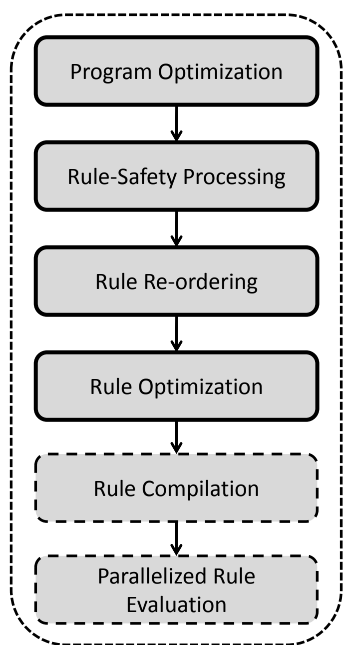

Figure 4.1: IRIS' Evaluation Pipeline. Components with dashed lines are concerned with parallel rule evaluation itself.

In order to process unsafe rules IRIS can be configured to use a rule augmentation processor. This processor uses a technique suggested by Gelder[34] that adds a 'universe' predicate for each unbound variable. This universe predicate automatically contains all term values that appear anywhere in the input program or that are created during program evaluation.

Rule Re-ordering Optimizations As discussed in previous section there can still be significant performance improvements if the rules are evaluated in a better order, i.e. rules that produce tuples that feed other rule bodies are evaluated earlier. The standard IRIS rule re-ordering optimizer searches for the first positive body predicate of each rule and builds a dependency graph between these positive body predicates and rule heads. Rules are then arranged following this directed graph.

Rule optimizations A number of optimizations can be achieved on a per rule basis. The default configuration contains the following four optimizers, but more user defined optimizers can be easily added.

Join condition This optimizer attempts to use the same variable for join conditions. For example

$$p(?X) : -q(?X), r(?Y), ?X = ?Y$$

would be changed to

$$p(?X) : -q(?X), r(?X)$$

This can significantly reduce the number of intermediate tuples produced during a sequence of cartesian products.

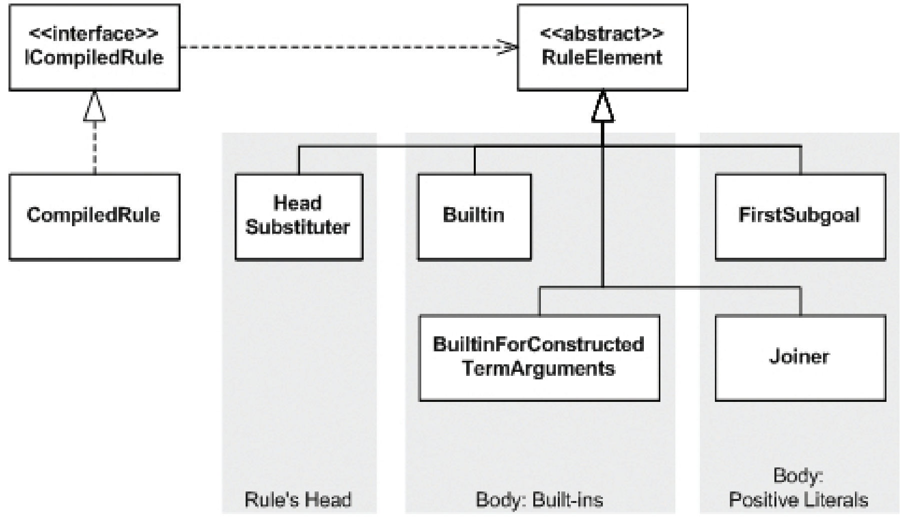

Figure 4.2: Rule elements in compiled rules.

Replace variables with constants This has the effect of pushing selection criteria into the evaluation of a relation, such that fewer tuples are processed, e.g.

$$p(?X, ?Y) : -q(?X, ?Z), ?Z = 2$$

would be changed to

$$p(?X, ?Y) : -q(?X, 2)$$

Re-order literals Re-arrange the literals in a rule body so that the most restrictive literals appear first. The preferred order is: positive literals with no variables, built-ins with no variables, positive literals, built-ins. However built-ins can be pushed earlier into the rule body as soon as all their variables are bound.

Remove duplicate literals Remove any literal that appears twice within the rule with the same variables.

Rule Compiler Compiling an input rule simply involves pre-computing information required for rule evaluation and constructing a specific object representation of a rule. The input rule is transformed into a compiled rule that can be quickly evaluated using a rule evaluator. For this purpose a compiled rule encapsulates all the required processing logic in dedicated *rule elements*. Possible rule elements are depicted in Figure 4.2.

Delegating the actual evaluation logic in this way to rule elements allows to use the same general evaluation scheme (e.g. naive or semi-naive evaluation) while easily changing the particular join algorithm used.

The first step during rule compilation is to create views on each literal. A view is analogous to a view in a relational database and is created from the underlying relation for a predicate and the tuple as it appears in the rule body predicate. A view is itself a relation for the purposes of rule evaluation, as the following examples demonstrate:

$$p(?X, ?Y) : -q(?X, ?Y), r(?Y, ?Y), s(1, ?X)$$

$q(?X, ?Y)$ is a simple view that selects all tuples from the relation for 'q'.
$r(?Y, ?Y)$ is a view that selects only those tuples where both terms are equal. This view appears as a unary relation.
$s(1, ?X)$ is a view that selects values from the second term of the relation for 's' where the first term is equal to 1. This view also appears as a unary relation. Thus, this compilation step can effectively filter the number of values that have to be considered in subsequent steps dramatically. The next step is to assign join objects and indexes. Since all joins in Datalog are natural joins, the compiling stage looks for all matching variables between two adjacent views, calculates the join indices and creates indexes. The indexes used in the default rule compiler are hash-based and therefore this approach is equivalent to performing a hash join. The advantages of a hash join over a sort-merge join are that the underlying views are not required to be sorted in any way, rather simply grouped according to matching join indices.

Rule Evaluator Several implementations of rule evaluators are already implemented within IRIS (e.g. naive and semi-naive evaluation). Rule evaluators generally accept a set of compiled rules and process them. However, as mentioned, the evaluation of an individual rule is encapsulated within implementations of the `ICompiledRule` interface. Available implementations can be re-used for that reason

The **NaiveEvaluator** loops through all rules and collects the results of feeding the already generated facts to the rule. The resulting tuples are stored in the relation of the corresponding predicate and then form part of the input to the next rule evaluation. Iteration over the rules is repeated until a pass does not produce any new tuples.

The **SemiNaiveEvaluator** introduces an important optimizations to this evaluation approach. In particular, it tries to avoid unnecessary processing of predicates and their facts. The sequence of events is as follows: First do a single pass over the rules of the stratum as with the NaiveEvaluator. Collect the changes and add themto the known facts. Then, deduce additional facts by iterating over the rules of the stratum, as long as the last pass brought new facts. This is similar to the naive approach, yet there are differences. For once, the input of one pass is only the previously unknown output from the pass before. Rather than using all facts available for each predicate, we take advantage of knowing that some facts have already been considered in the first pass and we only need to focus on the new. Thus, for each literal, we evaluate the entire rule but with the current literal only considering the deltas. This means that for each pass over the rule, one literal has the newly added facts as its foundation, while the other literals evaluate as normal.

4.2.1 Specific Extensions for Parallel Rule Evaluation

The modular architecture of IRIS, as outlined in the previous section, makes it possible to isolate the changes that enable the parallel evaluation of rules. Modular, individual components within IRIS are composed by a concrete *Evaluation Strategy* (e.g. a bottom-up evaluation strategy) which can be largely reused without changes. Since the actual evaluation of a rule (where parallelization takes place as explained in Section 4.1) is encapsulated within implementations of the `ICompiledRule` (see Listing 4.1) interface.

In order to implement the parallelization of joins we need to perform the following steps: i) Determines the required joins for a particular rules, ii) partitions the data

Listing 4.1: The `ICompiledRule` interface.

```
public interface ICompiledRule
{

  IRelation evaluate() throws EvaluationException;

  IRelation evaluateIteratively( IFacts deltas ) throws
     EvaluationException;

  IPredicate headPredicate();

  List<IVariable> getVariablesBindings();
}
```

in relations involved in the join (the map phase), iii) delegates joining the resulting subsets to lightweight reduce nodes, and iv) merges individual results and applies projection. Optionally it is possible to compute the optimal size of the shares for each join variable in step ii), as described in [1]. Thus the following concrete extensions are required:

1. A new `RuleCompiler` that analysis each rule and, instead of simply generating join objects, also computes *variable shares* and organizes an optimized data distribution according to the scheme in Section 4.1.

2. A new implementation of the `ICompiledRule` interface, that, when evaluation starts, uses this precomputed information in order to distribute join computation to remote hosts. Remote nodes can re-use the currently available implementation of a hash join algorithm in order to perform the actual join.

In this way *rule evaluators* that define the overall evaluation algorithm (naive, semi-naive, etc.) itself can remain unchanged. As a simple example for this behavior, the currently implemented naive evaluation algorithm is depicted in Listing 4.2.

Listing 4.2: Naive Evaluation Algorithm in IRIS.

```
public class NaiveEvaluator implements IRuleEvaluator
{

  public void evaluateRules( List<ICompiledRule> rules, IFacts
      facts, Configuration configuration ) throws
      EvaluationException
  {
    boolean cont = true;
    while( cont )
    {
      cont = false;

      // For each rule in the collection (stratum)
      for (final ICompiledRule rule : rules )
      {
        IRelation delta = rule.evaluate();

        if( delta != null && delta.size() > 0 )
        {
          IPredicate predicate = rule.headPredicate();
          if( facts.get( predicate ).addAll( delta ) )
            cont = true;
        }
      }
    }
  }
}
```

5. Conclusion and Future Work

In this document we initially examined the role of role rule-based reasoning on the Web and for LarKC in Section 2, and analyzed different parallelization approaches regarding their applicability for our specific case in Section 3.

Based on this, we described a mechanism that allows to efficiently compute joins required for arbitrary rules in a parallel and distributed manner in Section 4.1 by leveraging Map-Reduce as distribution scheme.

This approach can easily be integrated in our existing IRIS Datalog engine and we outline the architecture for a prototype in Section 4.2. This will result in the implementation of said strategy and further results in the future deliverable D4.4.2. This implementation will have the capabilities to evaluate rules in parallel by performing arbitrary joins in a distributed Map-Reduce fashion. This essentially results in a parallel, distributed, and scalable Datalog reasoner.

Equipped with a suitable RIF parser we can employ this engine as a parallel reasoner for RIF-Core rules along with RDF data. Such a RIF reasoner can be used along other standard reasoning components within LarKC, and thus adds further capabilities to the platform as a whole. Moreover, a reasoner for generic RIF rules can even replace "standard reasoners" for several tractable logical languages when used with suitable rule-sets. Thus, our reasoner can also be used as a parallel reasoner e.g. for OWL 2 RL or RDFS.

Potential future work includes a closer examination cost measurements in order to optimize data partitioning further. This also includes a more detailed examination of the trade-offs involved in different join evaluation strategies, see [25] for an initial overview. Moreover, dedicated heuristics and closer analysis of common join types (star joins, chain joins) could bring further performance benefits. More work on refined strategies for the treatment of highly recursive rules, that require a lot of Map-Reduce iterations, e.g. by allowing limited intermediate communication between reduce nodes, also seems a fruitful direction.

References

[1] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT '10: Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110, New York, NY, USA, 2010. ACM.

[2] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider. *The description logic handbook*. Cambridge University Press New York, NY, USA, 2007.

[3] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15. ACM, 1985.

[4] DA Bell, J. Shao, and MEC Hull. A pipelined strategy for processing recursive queries in parallel. *Data & Knowledge Engineering*, 6(5):367–391, 1991.

[5] B. Bishop and F. Fischer. Iris-integrated rule inference system. In *International Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008)*. Citeseer, 2008.

[6] D. Bitton, H. Boral, D.J. DeWitt, and W.K. Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Transactions on Database Systems (TODS)*, 8(3):324–353, 1983.

[7] H. Boley, G. Hallmark, and ed Kifer, M. RIF Core Dialect. *W3C Proposed Recommendation 11 May 2010*, 2010.

[8] Harold Boley and Michael Kifer. Rif overview. W3C Working Draft, May 2010.

[9] D. Brickley and R.V. Guha. Resource description framework (RDF) schema specification 1.0. *W3C Candidate Recommendation*, 27:2001–03, 2000.

[10] ed Dave Reynolds. OWL 2 RL in RIF. *W3C Working Draft 11 May 2010*, 2010.

[11] J. De Bruijn, H. Lausen, A. Polleres, and D. Fensel. The web service modeling language wsml: An overview. *The Semantic Web: Research and Applications*, pages 590–604, 2006.

[12] J. Dean and S. Ghemawat. Map Reduce: Simplified data processing on large clusters. *Communications of the ACM-Association for Computing Machinery-CACM*, 51(1):107–114, 2008.

[13] I. Horrocks et al. Swrl: A semantic web rule language combining owl and ruleml. In *W3C Member Submission*, volume 21, 2004.

[14] S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of Datalog queries. *ACM SIGMOD Record*, 19(2):143–152, 1990.

[15] B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. 2003.

[16] W3C OWL Working Group. OWL 2 Web Ontology Language: Document Overview. *W3C Recommendation*, 2009.

[17] G. Hulin. Parallel processing of recursive queries in distributed architectures. In *Proceedings of the 15th international conference on Very large data bases*, pages 87–96. Citeseer, 1989.

[18] G. Lukácsy and P. Szeredi. Scalable Web Reasoning Using Logic Programming Techniques. *Web Reasoning and Rule Systems*, pages 102–117.

[19] A. Miles and S. Bechhofer. SKOS Simple Knowledge Organization System Reference. *W3C Recommendation*, 2009.

[20] B. Motik, B. Cuenca Grau, I. Horrocks, Z. Wu, A. Fokoue, and editors C. Lutz. OWL 2 Web Ontology Language: Profiles. *W3C Recommendation*, 2009.

[21] B. Motik, U. Sattler, and R. Studer. Query answering for owl-dl with rules. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3:41–60, 2005.

[22] R. Mutharaju, F. Maier, and P. Hitzler. A MapReduce Algorithm for EL++. In *23rd International Workshop on Description Logics DL2010*, page 456, 2010.

[23] L. Raschid and S.Y.W. Su. A parallel processing strategy for evaluating recursive queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 412–419. Citeseer, 1986.

[24] RIF RDF and OWL Compatibility. Jos de Bruijn, ed. *W3C Proposed Recommendation 11 May 2010*, 2010.

[25] D.A. Schneider and D.J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the sixteenth international conference on Very large databases*, pages 469–480. Citeseer, 1990.

[26] M. Schneider, J. Carroll, I. Herman, and editors P.F. Patel-Schneider. OWL 2 Web Ontology Language RDF-Based Semantics. *W3C Recommendation*, 2009.

[27] J. Seib and G. Lausen. Parallelizing Datalog programs by generalized pivoting. In *Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 241–251. ACM, 1991.

[28] R. Soma and V.K. Prasanna. A Data Partitioning Approach for Parallelizing Rule Based Inferencing for Materialized OWL Knowledge Bases. In *Proc. of the 21st International Conference on Parallel and Distributed Computing and Communication Systems*, pages 19–25, 2008.

[29] R. Soma and VK Prasanna. Parallel inferencing for OWL knowledge bases. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 75–82, 2008.

[30] H.J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):79–115, 2005.

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

[32] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. Bal. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In *Proceedings of the 8th Extended Semantic Web Conference (ESWC2010), Heraklion, Greece*, 2010.

[33] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable distributed reasoning using mapreduce. *The Semantic Web-ISWC 2009*, pages 634–649.

[34] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM (JACM)*, 38(3):619–649, 1991.

[35] J. Weaver and J. Hendler. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. *The Semantic Web-ISWC 2009*, pages 682–697.

[36] O. Wolfson and A. Silberschatz. Distributed processing of logic programs. *ACM SIGMOD Record*, 17(3):329–336, 1988.

[37] H. Yang, A. Dasdan, R.L. Hsiao, and D.S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, page 1040. ACM, 2007.

[38] W. Zhang, K. Wang, and S.C. Chau. Data partition and parallel evaluation of datalog programs. *IEEE Transactions on Knowledge and Data Engineering*, 7(1):163–176, 1995.