TripCom

Triple Space Communication

FP6 – 027324

Deliverable

D3.3
Semantic matching in distributed spaces

Lyndon JB Nixon
Philipp Obermeier
Omair Shafiq
Janne Saarela
Vassil Momtchev

April 22, 2008

Executive Summary

The first Triple Space implementation handles semantic matching between knowledge tuples and knowledge templates in individual spaces, which exist wholly on one kernel (including subspaces). In other words, query answering takes place locally on a kernel. However, the vision of Triple Space is Web scale distribution of knowledge. We are extending Triple Space to support semantic matching as a distributed querying problem (where answers to subqueries may be on different kernels). Furthermore, self-organization of the knowledge will lead to distribution of even single spaces across kernels.

This leads to a need for new solutions to semantic matching which can provide answers based on the search for and combination of knowledge in different spaces, where performance and scalability are key issues. We also must be able to resolve inconsistencies in the underlying knowledge schema arising from the unavoidable partitioning of axioms across kernels, being provided by autonomous and independent clients.

As a preparation for our work in TripCom in developing a distributed semantic query tool which will support this functionality, this deliverable presents some initial solutions to the problem of semantic matching in distributed spaces. This forms a solid basis for the implementation work.

Following the state of the art in both areas, we propose a Query Preprocessing component which will handle distributed querying and inconsistent reasoning in the Triple Space kernel.

For the distribution of queries, we extend the state of the art and develop for the first time a cost model for SPARQL. Building on this, we propose "Adaptive Distributed Query Processing" which uses cost functions in the cost-based decomposition and concurrent distribution of queries over distributed triplespaces.

For inconsistent reasoning, we propose the use of the PION framework with the RIO reasoner for handling inconsistency arising when reasoning over knowledge in a Triple Space kernel.

We describe how both aspects will be implemented as part of the Query PreProcessor and present initial results from an example-based evaluation of both approaches. From this basis, we conclude with the outlook for supporting distributed semantic matching in a Web scale Triple Space.

Document Information

IST Project Number	FP6 – 027324	**Acronym**	TripCom
Full Title	Triple Space Communication		
Project URL	`http://www.tripcom.org/`		
Document URL			
EU Project Officer	Werner Janusch		

Deliverable	**Number**	3.3	**Title**	Semantic matching in distributed spaces
Work Package	**Number**	3	**Title**	Triple Space Interaction

Date of Delivery	**Contractual**	M24	**Actual**	31-March-08
Status	version 1.0		final ☒	
Nature	prototype ☐ report ☒ dissemination ☐			
Dissemination Level	public ☒ consortium ☐			

Authors (Partner)	Lyndon JB Nixon			
Resp. Author	Lyndon JB Nixon		**E-mail**	nixon@inf.fu-berlin.de
	Partner	FUB (Free University of Berlin)	**Phone**	+49-30-838 75225

Abstract (for dissemination)	This deliverable presents the Query PreProcessor component as a solution to semantic matching in distributed spaces. The solution is based on (1) adaptive distributed query processing and (2) inconsistency detection in the Triple Space. Introduction, concept, proposed solution, implementation and evaluation plans as well as examples have been presented for both of the solutions.
Keywords	Semantic, Distributed, Query Processing, Inconsistent Reasoning

Version Log			
Issue Date	**Rev No.**	**Author**	**Change**
2007-09-26	1	Lyndon Nixon	First draft structure
2007-10-20	2	All Partners	Updated their sections
2007-12-03	3	Omair Shafiq	Restructured the draft structure
2008-02-19	4	Lyndon Nixon	Restructured again
2008-02-20	5	Omair Shafiq	Input in Querying Requirements
2008-02-29	6	Omair Shafiq	Input in Inconsistency Reasoner in overall framework
2008-03-07	7	Omair Shafiq	Updated using PION in TripCom in Query Add-ons
2008-03-12	8	Omair Shafiq	Updates in Query Implementation
2008-03-14	9	Omair Shafiq	Updates in Query Results
2008-03-15	10	Omair Shafiq, Philipp Obermeier	Review and updates
2008-04-09	11	Lyndon Nixon, Philipp Obermeier, Omair Shafiq	Review and updates in all chapters
2008-04-15	12	Lyndon Nixon, Philipp Obermeier, Omair Shafiq	Refined abstract and keywords

Project Consortium Information

Acronym	Partner	Contact
Semantic Technology Institute Innsbruck `http://www.sti-innsbruck.at`	STI	Prof. Dr. Dieter Fensel Semantic Technology Institute (STI) Innsbruck, Austria E-mail: dieter.fensel@sti-innsbruck.at
National University of Ireland, Galway `http://www.deri.ie`	NUIG	Dr. Laurentiu Vasiliu Digital Enterprise Research Institute (DERI) Galway, Ireland Email: laurentiu.vasiliu@deri.org
University of Stuttgart `http://www.iaas.uni-stuttgart.de/`	USTUTT	Prof.Dr. Frank Leymann Inst. für Architektur von Anwendungssystemen (IAAS) Stuttgart, Germany E-mail: frank.leymann@informatik.uni-stuttgart.de
Vienna university of Technology `http://www.complang.tuwien.ac.at/`	TUW	Prof.Dr. eva Kühn Institut für Computersprachen Vienna, Austria E-mail: eva@complang.tuwien.ac.at
Free University Berlin `http://www.ag-nbi.de/`	FUB	Prof. Dr.-Ing. Robert Tolksdorf AG Netzbasierte Informationssysteme Berlin, Germany E-mail : tolk@inf.fu-berlin.de
Ontotext Lab, Sirma Group Corp. `http://www.ontotext.com/`	ONTO	Atanas Kiryakov, Vassil Momtchev, Ontotext Lab, Sirma Group Corp. Sofia, Bulgaria E-mail: vassil.momtchev@ontotext.com
Profium OY `http://www.profium.com/`	Profium	Dr. Janne Saarela Profium OY Espoo, Finland E-mail: janne.saarela@profium.com
CEFRIEL SCRL. `http://www.cefriel.it/`	CEFRIEL	Davide Cerri CEFRIEL SCRL. Milano, Italy E-mail: cerri@cefriel.it
Telefonica I+D `http://www.tid.es/`	TID	Noelia Pérez Crespo Telefonica I+D Madrid, España E-mail: npc@tid.es

Table of Contents

1 Introduction

Aside from the emission of knowledge in the form of RDF triples into the Triple Space, the most common form of access to the Triple Space will be the rd operations of the TS API. These operations allow clients to receive RDF triples in a space as a response to some query made.

We have discussed Triple Space querying in D3.2 [19] and identified SPARQL as the best basis on which to implement the API rd operations. To phrase this differently, the TS API (see the revised API in D6.3) foresees an abstract Template object to represent a Triple Space query. In the core API, we support rd with a SimpleTemplate object which is seen as a subclass of Template, and which represents the simplest query possible for a highly scalable Triple Space. In the current implementation, in combination with the work on distributing the Triple Space (which will be described further in D2.4 [2]), we support Templates which contain SPARQL queries, while the SimpleTemplate is seen as being a single triple pattern (no conjunction or disjunction). In the distribution work, we have noted the negative impact on scalability once queries in a distributed system begin to include joins (conjunctive queries) and other constructs between query results from different spaces.

For the extended API, which will be the minimum API to be supported by our scenarios, we must support more complex querying, ideally full SPARQL over distributed knowledge. In order to still examine the potential of supporting subsets of SPARQL more efficiently by optimizing certain aspects of query handling as well as extending the Triple Space to support extensions to SPARQL by taking advantage of the internal implementation (e.g. that the reasoner chosen by Triple Space supports a richer logical model than RDFS), we need a solution which is independent of the underlying reasoning and storage layer. Even if we focus purely on the aspect of querying with SPARQL, the introduction of the distribution of semantic data in Triple Space raises new problems. While the solution for the first implementation is to pass the full SPARQL query to a space to be locally resolved by the kernel which manages it, this prevents the possibility of answering a query which is only possible through the combination of answers to parts of the query which are on different kernels. Furthermore, to more efficiently distribute knowledge at a Web scale, we will introduce self-organization principles where a space may be distributed across several kernels, so that even a query on a named space may require query resolution on several kernels.

To enable the final Triple Space implementation to support distributed querying in an efficient and scalable manner we present a solution based on a SPARQL cost model. As no previous work has developed a SPARQL cost model, this work represents a significant contribution of this deliverable.

Considering existing approaches based around query optimisation, we note that there are solutions in the distributed database field which could be applied with some adaptation to SPARQL, which is built upon the relational database algebra. Hence we present a query answering solution based on query cost estimation, decomposition and distributed resolution. As this work requires a significant theoretical basis, we concentrate here on the proof of the aspect of query cost modeling and present a concept for using query cost functions to decompose queries and handle them in a distributed fashion. This concept will be implemented in the final Triple Space implementation.

Furthermore, the distribution of knowledge in Triple Space applies also to the underlying ontologies with which the knowledge is reasoned over. These ontologies are available at the individual kernels and, since Triple Space does not attempt full synchronization of available ontologies, this can lead to inconsistencies in axioms on different kernels. This must also be resolved for the query answering, hence we also consider solutions where inconsistency is detected when performing reasoning.

The work on Triple Space scalability has led to certain decisions regarding guarantees of correctness and consistency. It is recognized that across distributed triplespaces, to preserve scalability it is unreasonable to expect a constant synchronization between knowledge in different spaces, nor that all the required knowledge to answer a query is findable or should be sought in the distributed spaces. Hence, in query answering, we must consider that (a) knowledge found in different spaces may not be consistent with one another and (b) knowledge being reasoned over may not be consistent with regard to the schema information that can be retrieved. As a result we plan for a component as part of our Query PreProcessor which will support inconsistent reasoning. In this deliverable, we present a concept for integrating such a component into the semantic matching function of the Triple Space and a prototypical implementation of such a component using the PION Framework [36].

Both distributed query answering and inconsistent reasoning support will be implemented as part of the Query PreProcessor component, which handles queries before being resolved at the TS Adapter (reasoner) or the Distribution Manager. For the handling of SPARQL queries in the current implementation, where we use the Distribution Manager to select a target space in the distributed system and send the entire SPARQL query for resolution at that space, please refer to D2.4 [2].

2 State of the art

This chapter presents the state of the art in existing approaches for distributed querying and inconsistent reasoning support. To develop a concept for semantic matching in distributed triplespaces, we consider our contribution to the state of the art in distributed semantic querying, focusing on SPARQL, and inconsistent reasoning.

2.1 Current querying support

Current querying support is provided in Triple Space by the ORDI Framework. Reasoning is provided by TRREE 2.0 which is implemented in OWLIM. This implementation provides support for a subset of DL called OWL Horst [33]. OWL Horst adds support for typed literals and partial support for OWL primitives like FunctionalProperty, TransitiveProperty and disjointWith. OWL Horst imposes no constraints on the RDFS semantics and makes possible decidable rule extensions of OWL. OWLIM reasoning supports different inference modes, including rdfs (the standard RDFS semantics), partialRDFS (an optimisation of RDFS support) and owl-horst (an extension with rules and DL support). OWL support in OWLIM avoids DL-specific constraints such as metamodelling, hence allowing RDFS semantics to be respected. RDFS support has only a single restriction on typed literals for performance optimisation. The partialRDFS mode improves RDF-based inference by:

- excluding some ”trivial” RDFS axioms
- not inferring all concept types

With partialRDFS, the LUBM benchmark queries are evaluated properly. Hence the current reasoning support in Triple Space should be able to provide semantic matching, including full SPARQL query support, at least locally on a single kernel. When we consider that Triple Space knowledge will be distributed across spaces on different kernels, we must ask:

- What are the problems in semantic matching in distributed spaces?
- What are the approaches to do semantic matching in distributed models?
- Why we choose one of them?

The query engine and the semantic matching are typically represented like that in the triple stores:

- 1. RDF data model (explicit data) - this layer understands only how to store and read the data; possibly to generate some data statistics
- 2. Reasoner to interpret semantics (implicit + explicit data) - this layer has to derive new implicit data query or write time
- 3. Query engine - this layer know how to translate the query language to conjunction of triple patterns; possibly the use of information from the underlying layers to optimize the query;

There is no need to re-implement the RDF data model or reasoner, as these are provided in WP1. However, the provided implementation is designed to provide reasoning support over a local knowledge base, i.e. answering different parts of SPARQL queries with knowledge from different knowledge bases is not supported. Also, the reasoner assumes complete and consistent knowledge in the knowledge base for its inference calculations. Hence, we turn to these two open challenges for TripCom semantic matching in distributed spaces: how to distribute SPARQL queries and how to handle querying inconsistent knowledge bases.

2.2 State of the art: (Distributed) SPARQL querying

In the last years, the query language SPARQL has evolved into a widely accepted standard for querying RDF. In January 2008 it was officially approved by a W3C Recommendation[29] – a status which no other RDF query language has achieved. A respectable number of stable and performant SPARQL query engines have been developed, e.g. Jena ARQ, the Sesame SPARQL plugin or OWLIM as part of ORDI. However, the consideration of optimizations to SPARQL query answering is still at an early stage of development, despite much relevant work already having been done in the distributed databases field. This work is applicable to SPARQL as the semantic query language is similar in its relational algebra to SQL. However, current solutions have focused on optimisation within the reasoner e.g. by more efficient indexing of RDF. By developing a cost model for SPARQL, we propose in contrast to this an optimisation layer over the reasoner and hence independant of individual choices of storage or reasoning. We consider this a more applicable solution in TripCom, as individual kernels may choose different TSAdapter component implementations.

Some breakthroughs in the analysis of the expressiveness and classification of SPARQL with respect to other query languages has been achieved by Perez et al.[26]. In their work they determined the complexity of certain types of SPARQL expression and developed an algebra for SPARQL and SPARQL graph pattern expressions which was a significant contribution to the theoretical algebraic model *SPARQL Abstract Queries* of SPARQL queries. Except for some minor exceptions the possibility of a straightforward reduction of SPARQL to a SQL-resembling relational algebra has been shown in [9] which is the theoretical core of several SPARQL-to-SQL transforming SPARQL-DBMS such as D2RQ or SquirrelRDF. Additionally it can be attributed to the similarity between SPARQL and SQL algebras that a System-R-like graph model for SPARQL query plans together with optimization techniques query plans could be introduced [17].

Since many Semantic Web applications eo ipso manage and store data in distributed places, an optimization of distributed SPARQL query processing is necessary. Development is in a premature state since none of the widely used query engines, including the ones mentioned above, are designed for distributed query processing at all. Research in distributed SPARQL query processing started recently and predominantly leans on optimization techniques for general database query languages and other RDF query languages. Heiner Stuckenschmidt et al. [32] suggest for querying distributed RDF-repositories a schema-path-based indexing and storage organization as well as heuristics for join ordering. The techniques are implemented as a prototype extension for Sesame. In [7], *RDFPeers*, a spanning of RDF-Repositories over a Peer-to-Peer network is introduced. Execution of disjunctive and conjunctive queries over the distributed storages are described though RDFpeers doesn't support SPARQL as query language. In contrast to that Andreas Harth et al. [16] propose *YARS2*, a toolkit for efficiently querying distributed storages of graph-shaped data, especially RDF data and SPARQL are supported. A distributed indexing mechanism for quadruples (representing RDF triples) and distributed SPARQL query processing with join ordering are fundamental contributions of this work. Though while lookups for indexes can be dispatched concurrently in the distributed storage environment, other operations embodied in a SPARQL query (i.e. *Join*, *Union*, etc.) are solely executed at one host in the system. An adaptive cost estimation based on System-R-like physical cost and cardinality functions is presented in [30] for conjunctive OWL Reasoning. There is no SPARQL support, rather queries are mapped to a set of deductive rules which are processed in a Datalog engine. Additionally, there is no known implementation of the work.

In contrast to SPARQL the opportunities offered by distributed query processing are vastly explored in the database community. Kossmann describes in [20] a series of query processing optimization concepts in distributed databases in a textbook style. Pirahesh et al.[28] present rule-based rewriting techniques for queries which deliver optimization on a logical level independent from the physical implementation. Query plan cost models which sum up physical costs for single operators are suggested in [31, 21]. A sophisticated survey of adaptivity for query processing, especially intra-query adaptivity, was assembled by Deshpande et al. [10]. Adaptivity means that the query processing is interleaved with adjustments by the query optimizer.

A cost model-based technique for query adaptation as described above in the distributed databases field are not available yet in a distributed SPARQL processing context. A cost model is also a pre-requisite in a subcomponent for a query optimizer for distributed SPARQL processing, to serve as an indicator for decision making of other subcomponents such as query rewriting, decomposition or join and selection ordering. As no SPARQL cost model is currently available, a contribution of this work has been to develop one as part of a Query PreProcessor component in the Triple Space kernel.

2.3 State of the art: Inconsistent reasoning

A classical example of inconsistency in ontologies can be seen as, when a new ontology statement is specified, the new statement has to be checked carefully that whether it is consistent not only with respect to already existing rules, but also to the rules that can be added in future. This is a design issue and is difficult for a knowledge engineer to ensure and maintain the consistency in ontology specifications at a large scale. Lets consider a situation where two rules are added in an animal ontology, i.e.

- Birds are animals
- Birds are flying animals
- Animals are either flying animals or not flying animals

One can realize that birds can fly but this is not valid in general. Lets assume that at a later stage the ontology is further expanded by adding following new rules:

- Eagles are birds
- Penguins are birds
- Penguins are not flying animals

The newly introduced concept penguin in the ontology of animals becomes unsatisfiable, because according to the rules of the ontology one can deduce that penguins are both flying and not flying animals. Yet penguins are also animals, and animals can either only fly or not fly. That leads to an inconsistent ontology.

Triple Space is an open, distributed and heterogeneous middleware which is used by multiple clients communicating with each other regardless of time, space and reference. It is on the one hand good that it helps in complying with the principles of the Web and keeps the communication simple for the globally distributed clients independent from each other. On the other hand, it can also cause inconsistency in knowledge when it is being manipulated by multiple users without any synchronization with each other. There are different reasons why inconsistency may arise. Normally in a Web-based setting, different users are adding and altering ontological information without any synchronization of time, reference, location and any background/context. The reasons identified for inconsistencies can be due to the ambiguous representation of rules and can occur if new statements are added to the ontology which are not consistent with old rules. Usage of terminologies that have dual or multiple meanings as well as migrating ontologies from one data source to another data source can also cause inconsistencies in ontologies. There are two main ways to deal with inconsistency in ontologies. One is to diagnose and repair it when we encounter it. However, the other approach is to simply avoid the inconsistency and to apply a non-standard reasoning method to try to obtain meaningful answers. One solution in this approach could that we simple remove the contradicting rule yet one may not be sure which part of the ontology should be removed. Another solution could be that we divide the ontologies into different parts and maintain their local consistency, but this would not allow the ontologies to evolve.

There are some related approaches that exist for reasoning with ontologies at a large and distributed scale where possibilities for arising inconsistencies exist. An approach for reasoning with inconsistent ontologies in Peer-to-Peer Inference Systems [27] generates the possible positive and negative consequences to be analyzed whenever a new rule is added in an ontology. Moreover, the PION framework [36] for reasoning on inconsistent ontologies is another kind of coherence-based approach where, for a given query, it aims at finding a specific consistent subset of the global theory, in which the query (or its negation) classically holds. If it is not possible, the answer is undetermined. The set is constructed by successive consistent expansions, according to some selection function measuring some relevance criterion with the query. Along with all these solutions, Epistemic semantics based on Epistemic logics has been proposed to deal with possible inconsistencies in P2P Data Inference Systems formalized in a first order multi modal language [11]. It considers the case of local inconsistency as well as global inconsistency. Mappings to cover the consistencies are formalized in such a way that they cannot be used to propagate information from some locally inconsistent theory. Moreover, mappings can only be used to propagate information to a peer, as far as they do not contradict either local information or other non-local information that may be deduced on that peer.

In the sections below we further review the two most promising approaches for handling inconsistency, Defeasible Logic and the PION Framework. We have finally chosen the PION Framework as a basis for the inconsistency reasoning solution in Triple Space because (1) this solution is suitable to target open (web-like) environments like Triple Space (as it will find consistent subsets of ontologies on kernels), (2) is comparatively scalable with a growing amount and size of ontologies as well as instance data, and (3) is implemented, evaluated [35] and available as open-source [1] to be used and enhanced further according to Triple Space requirements.

2.3.1 Defeasible Logic and Reasoning

In classical mathematical logic, e.g. propositional, first-order or second-order logic, reasoning in deductions systems is *monotonic*, i.e. a statement inferred by a deduction could not be revised by adding any new formulas to the set of axiomtic formulas the deduction is based on. The concept of monotonic reasoning is very convenient and established but obviously not sufficient for scenarios where some but not all information is available. As a matter of fact human reasoning is most of the time *non-monotonic*, i.e. based on inconsistent information: We often reject old conclusions based on new evidence, even when those old conclusions where justified at the time we arrived at them. We also will face situations where we must draw conclusions with only a partial knowledge of our world. On a theoretical level non-monotonic reasoning could be achieved by a *Defeasible Theory* which is basicly an aggregation of a "defeasible" deduction system and axioms described in the language of *Defeasible Logic* [22]. In Defeasible Logic formulas are always literals of a logic language, e.g. atoms and negated atoms of the propositional language. A *defeasible theory* is a quadruple $(F, R, C, \prec)$ such that F is a set of formulas, R is a set of *rules* consisting of *strict rules*, *defeasible rules* and *undercutting defeaters*, C is the set of *conflicting formulas* and $\prec$ is an acyclic binary relation on the non-strict rules in R. The literals in F represent our a priori knowledge, in logic programming often called *facts*. *Strict rules* are inference rules in the classical sense: When ever the premises are indisputable then so is the conclusion. *Defeasible rules* are rules which can be defeated by contrary evidence. That is either an assertion inferred by a strict rule respectively by a superior defeasible rule according to $\prec$ or an *undercutting defeater*. An undercutting defeater implies a literal from already deducted knowledge for the only purpose to prevent a defeasible rule application. Within a defeasible theory we can *defeasibly deduct/proof* a proposition by applying the rules in R according the the description above. Defeasible Reasoning is not complete but sound.

Defeasible Logic Programming - DR-Prolog

There exists several implementations of Defeasible Logic as d-Prolog or DR-Prolog [3]. Both are defeasible logic extensions of Prolog and are based on a mapping P from Defeasible Theories into Prolog-Programs such that the following theorem applies:

> p is defeasibly provable in a defeasible theory $D \Leftrightarrow$
> p is included in all stable models of $P(D)$.

For our purpose DR-Prolog seems to be a good choice because it was tailored to fit the needs of a Semantic Web context: The system can reason with rules and ontological knowledge, through the transformation of the RDFS constructs and many

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

OWL constructs into rules. Furthermore it is compatible with RuleML. In general, the performance of DR-Prolog is proportional to the size of the problem. This is because the defeasible theories are translated into logical programs with the same number of rules. DR-Prolog performs better in the case of theories that contain strict rules, as in these cases the system has to process a smaller number of rules in order to find the appropriate conclusion. Comparing to d-Prolog, DR-Prolog performs better in the cases of complex theories. Especially in the case of theories with disputed inferences, d-Prolog performs very badly, with time growing exponentially in the problem size. d-Prolog is substantially more efficient than DR-Prolog when there are only strict rules, due to the direct execution of such rules. However, d-Prolog shows its incompleteness, comparing to the other two systems, when it loops on cyclic theories.

2.3.2 PION Framework

PION [34] is a framework for processing inconsistent ontologies that has been proposed to enable the returning of meaningful answers to queries where the reasoned ontologies are inconsistent. They find four different ways as to how inconsistency can occur in the Semantic Web:

- Inconsistency by misrepresentation of default rules
- Inconsistency caused by polysemy
- Inconsistency through migration from another formalism
- Inconsistency caused by multiple sources

When a knowledge engineer specifies an ontology statement, in particular a rule, she/he has to check carefully that the new statement is consistent, not only with respect to existing rules, but also with respect to rules that may be added in the future, which of course may not always be known at that moment. This makes it very hard to maintain the consistency in ontology specifications and can lead to ”Inconsistency by misrepresentation of default rules”. There can be the case when some words are used in the ontology as concepts that have multiple meanings. This phenomenon is called as Polysemy. This is what leads to the ”Inconsistency caused by polysemy”. When the specification of an ontology is moved or migrated from other data sources, there are possibilities for inconsistencies to occur. This kind of inconsistencies are called as ”Inconsistency through migration from another formalism”. An ontology specification developed from multiple sources naturally brings inconsistencies. This can be a result of different activities, i.e. merging, aligning or integrating separate ontologies. The Solution is not restricted to ontology specifications of a particular formalism. There are several formal definitions that have been proposed regarding inconsistency reasoners which are listed and explained briefly as follows:

- Soundness: an inconsistency reasoner should be considered correct if the formulas that follow from an inconsistent theory X follow from a consistent subtheory of X using classical reasoning.
- Meaningfulness: the output of an inconsistency reasoner is meaningful if and only if it is consistent and sound. Namely, it requires not only the soundness condition, but also all of the answers provided by the reasoner are meaningful.

- Local Completeness: Global completeness is impossible due to inconsistencies.
- Maximality: An inconsistency reasoner is maximal if and only if there is a maximal consistent sub-theory such that its consequence set is the same as the consequence set of the inconsistency reasoner.

PION framework is a framework for reasoning with inconsistent ontologies. It is based on the approach to simply avoid the inconsistency and to apply a non-standard reasoning method to obtain meaningful answers as is more suitable in a Web setting. The PION framework has been evaluated by investigating its different selection functions and testing them on large scale and realistic ontologies. The first selection function that has been chosen is based on finding the syntactic connections between axioms in order to decide which axioms are relevant to a query. This relevance requirement is decreased gradually during reasoning in order to obtain a increasing subset of the axioms until it has selected a subset which is small enough to avoid the inconsistency, but large enough to answer the query. There is a second selection function that takes into account that it is dealing with ontologies and uses the concept hierarchy for the selection of related axioms. The main idea behind the design of PION framework is, given a selection function, a consistent subtheory from an inconsistent ontology is selected. Then a standard reasoning on the selected subtheory is applied to find meaningful answers. If a satisfying answer cannot be found, the relevance degree of the selection function is made less restrictive thereby extending the consistent subtheory for further reasoning.

Since Triple Space is foreseen as being an environment similar to the Web, with autonomous clients acting in a loosely coupled manner on knowledge stored on individual kernels which make up in their totality the distributed system of Triple Space, ee consider the PION framework the best candidate for inconsistent reasoning support in TripCom.

3 A Query PreProcessing component

3.1 Query PreProcessor Overall Concept

The Triple Space is a distributed system consisting of nodes, called *Kernels*, with heterogeneous hardware resources and software environment (OS, applications). Each kernel provides a client API (based on the Linda coordination language) and a query processor in the underlying storage layer (implemented with OWLIM and the ORDI framework). The system should scale for a number of kernels proportional to the size of the Internet. Therefore it is evident that we require a query processing which scales not only vertically (i.e. with increasing local resources in a kernel) but also horizontally (i.e. by adding new kernels). Additionally, as the number of kernels increase, the distribution of the ontologies used for the reasoning will become thinner, increasing the opportunity for inconsistencies to arise as contradictory axioms are introduced in individual kernels and yet brought together when knowledge is clustered for a distributed reasoning task.

Such scalable and consistency preserving query processing capabilities will be provided by the Query PreProcessor. The Query PreProcessor (Fig. 3.1) consists of two subcomponents, a Query Optimizer and an Inconsistency Reasoner. During the processing of the rd instruction (a semantic query in Triple Space) the Query PreProcessor communicates in both directions with the Distribution Manager and the TS Adapter.

The purpose of the Query Optimizer, sitting on top of the SPARQL query processor, is the extension of the "locally working" SPARQL query processing of each kernel to support adaptive distributed operation (support for distribution querying and adaptation to maximize the efficiency of answer acquisition). The purpose of the Inconsistency Reasoner is to ensure that the results obtained from the local query processor in the kernel are based on a consistent logical theory (the ontology), proofing the ontology to detect and avoid any inconsistencies as a first step. It ensures that there is no inconsistency in the results obtained at the local query processor before they are returned to the client.

3.2 Query PreProcessor Interface

The Query PreProcessor takes the following type of tuples as input from the Distribution Manager

```
TemplateEntry tmp, URL space, Time ts, Enum kind,
ClientInfo clientInfo, int opId, URL transId, Set<URL> subspaces,
QueryProcessingData data, boolean toFromReasoner
```

The Query Pre-Processor send the following type of tuples (RdDMEntry) to the Distribution Manager

```
TemplateEntry tmp, URL space, Time ts, Enum kind,
ClientInfo clientInfo, int opId, URL transId, Set<URL> subspaces,
QueryProcessingData data, boolean toFromReasoner
```

The input and output entry types have two components, `QueryProcessingData data` and `boolean toFromReasoner`, which are exclusively introduced for the Query PreProcessor. The `QueryProcessingData` class contains either the data needed by the

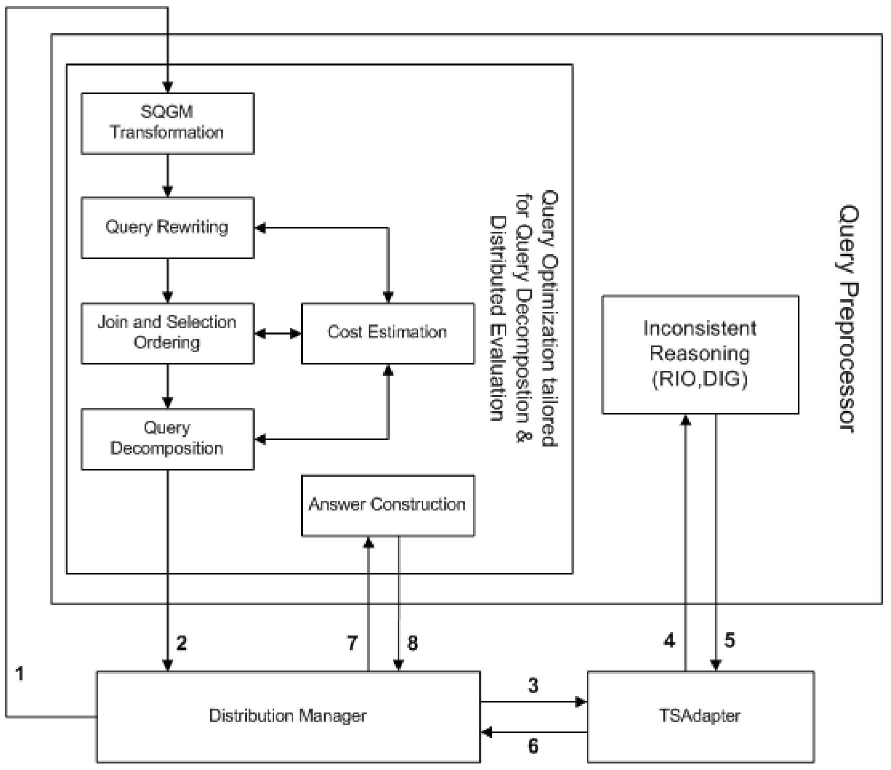

Figure 3.1: Architecture of the Query PreProcessor subcomponents

query optimizer or the inconsistent reasoner. The query optimisation data is provided by the distribution manager to guide the adaptive distributed query processing. In our concept, this is a list of kernels able to answer parts of the query together with information about their respective processing loads (see 4.2.2 for more information). The inconsistent reasoner data is provided by the TS Adapter to support the inconsistency checking. In our concept, this is a subset of the logical theory used by the reasoning task to acquire the query results. The flag `boolean toFromReasoner` determines if input/output data is directed to/coming from the Query Optimizer or Inconsistent Reasoner, which are working highly independently as the two top-level components of the Query PreProcessor.

3.3 Query Optimizer Overall Concept

The query optimizer in combination with the Distribution Manager facilitates the query optimization for our 'query decomposition and distributed evaluation' approach. While for this matter the task of the Distribution Manager primarily lies in the fast lookup of target knowledge sources, dispatching queries to other kernels and retrieval of solutions, the Query Preprocessor provides the modules for query rewriting, cost estimation, join & selection ordering and query decomposition & answer construction. The innovative cost-based decomposition paradigm which is showcased in Section 4 yields the possibility of concurrent query evaluation in the Triple Space. Thus it is a fundamental concept to guarantee efficiency in distributed query answering and hence scalability.

After the Query Optimizer has deconstructed a query plan using the physical parameters from the Distribution Manager, it determines which kernels perform which subqueries and gives the respective subquery plans to the Distribution Manager for forwarding. This forwarding is predicated on a determination of query cost and physical parameters, so that the most efficient path for answering the query is chosen. As a result we support a distribution of subqueries oriented on evaluation costs and available resources, introducing a novel new paradigm which not only fits the requirements of Triple Space but could have application in other distributed RDF scenarios.

Query processing is performed in an *adaptive intra-query* manner [10] which means that the query processing task is divided into a sequence of cycles. In each cycle a segment of the whole query plan is processed as if itself would be a query-plan on its own, i.e. optimized by the optimizer, forwarded by the Distribution Manager and executed by the query processor(s), with respect to the results generated by the preceding cycle. The first cycle starts with triple patterns in the query plan, and going from there deals with the sub graph of the query plan reachable in a directed Breadth-First-Search traversal taking the triple patterns as roots. This traversal is limited by a maximum path length n between a triple pattern and its reachable nodes to limit cost estimation errors in the query optimization phase. The second cycle than uses the results of the first cycle and in the same fashion deals with the next query plan segment, and so on. In the following referred to these cycles as *adaptivity loops*. This kind of sub-division of the query processing task compared to an upfront optimization for the complete query plan, moderates the exponential error growth of the evaluation cost and solution cardinality estimation, which is crucial for an effective query decomposition and distribution.

3.4 Inconsistent Reasoner Overall Concept

The Inconsistency Reasoner checks if inconsistencies arise in the ontologies used in a reasoning task and ensures that consistent results are obtained by the local query processing.

Query results from a TS Adapter component are published in the system bus of the Triple Space kernel for the Query PreProcessor. Here the Inconsistency Reasoner comes into play and performs a check for any possible inconsistencies in the results obtained. In case of any inconsistencies, a sub-set of the result set is obtained that does not have any inconsistency with respect to the ontologies in the Triple Space locally accessible via the TS Adapter. After performing the inconsistency check, the local query result is ready for the Query PreProcessor to send back to the Distribution Manager.

To perform the inconsistency check, a specialized reasoner, called RIO, is used. When the query result is ready, the control component of RIO [34] processes the results together with the retrieved ontology information. RIO analyzes the retrieved data and based on its selection functions finds and resolves any inconsistency by selecting a consistent subset of the ontology. The query may be performed again after the removal of inconsistency in the ontology, so that answers may be acquired which respect a consistent underlying logical theory. Once both a consistent subset of the ontology is determined and a set of valid results for the query, the results are returned.

4 Proposal for Adaptive Distributed Query Processing

4.1 Introduction

In this document we propose a concept for distributed adaptive query processing in Tripcom. It inherently banks on two components embodied in each kernel: A SPARQL query processor and a query optimizer extending this processor such that queries can be processed in an adaptive distributed manner spanned over the kernels in the Triple Space.

More precisely, the optimizer will facilitate horizontal scalability for query processing (i.e. the query processing resources in Triple Space should scale proportional to its number of kernels/hosts) as an autonomous component independent of the query processor implementation used locally in a kernel. The sub techniques of the optimizer — cost estimation, query rewriting, join/selection ordering, query decomposition — are applied *adaptively*, i.e. they interleave query processing for dynamic adjustments that provides better response time or more efficient usage of resources (CPU or I/O). Thereby cost estimation has a essential role in the query optimizer since it should act as an indicator for other components for decision-making. Since the relational database community vastly explored the field of distributed querying we as much as possible rely on scientific results of the Relational Databases domain [20, 10, 8] and adapt these to our conditions. For the modelling of SPARQL queries on a logical level we use *SPARQL Query Graph Models (SQGM)* [17] which are graphs which exactly correspond to the algebraic formalization of SPARQL queries, *SPARQL Abstract queries* (see chapter 12 in [29]).

Due to the fundamental role of the estimation of SPARQL query evaluation costs we present in the following chapter a cost model for this purpose in detail. Consequently we declare conceptual ideas for the other sub components whose elaboration is planned for the future subsequent task.

In Section 7.1 the benefits of a query decompostion concept which makes its decision based on our costmodel is demonstrated. Such a cost-based decomposition yields the possibility of concurrent query evaluation in the Triple Space System. Thus it is a fundamental concept to guarantee scalability.

4.2 Cost Model for Querying Distributed RDF-Repositories

In this section we present a cost model for the evaluation of SPARQL queries tailored to act as part of an optimizer for adaptive, distributed query processing [10, 20] supporting other subcomponents like query decomposition, query rewriting and join order selection for cost-based decision-making. Our cost model is similar to the ones for System-R suggested in [31, 21] in that it estimates physical costs for each individual operation of a query and then totals these costs. Furthermore costs are distinguished by the type of used resources, i.e. number of CPU instructions, number of I/O operations (page/read accesses on persistent memory) and net delay for retrieval of remote data. More precisely, the cost model poses a recursive cost function for query execution based on physical cost and cardinality functions for basic operations in SPARQL queries interpreted as *SPARQL Abstract Queries* [29], a declarative representation of

SPARQL queries (see section 4.2.1). Furthermore we presume a SPARQL query resp. query plan is given as a *SPARQL Query Graph Model* [17] which is a graph model for queries abstracting from the semantic of the corresponding SPARQL Abstract Query (see section 4.2.1). For the choice of complexities of basic SPARQL operations we argue that complexities of best-practice algorithms for related operations in the relational algebra for RDBMS can be used as an orientation.

4.2.1 Preliminaries

Formalization and Semantics of SPARQL Queries

SPARQL Abstract Queries For the study of SPARQL semantics, [29] provides a declarative formalization for a SPARQL query, called *SPARQL Abstract Query.* A SPARQL Abstract Query is a triple that consists of a *Query Form (QF)*, a *SPARQL Algebra Expression* and a *RDF Dataset.*

Query Forms are a set of functions $\{Select, Ask, Construct, Describe\}$ which take a *solution mapping multi-set* (see Definition 1) as input, $Construct$ additional uses a set of tripe pattern templates. $Select$ returns a set of variable bindings, Ask a boolean, $Construct$ and $Describe$ a RDF graph.

The SPARQL Algebra is an algebra representing graph patterns and solution modifiers in queries. It provides a set of unary or binary functions $\{Join, Union, Diff, LeftJoin, ...\}$ and a set of constants, called *Basic Graph Patterns (BGP).* Both function and constants return a solution mapping multi-set as result while taking solution mapping multi-sets and logical boolean constraints for variable bindings as input. In the following we refer to the set of these functions as *Sfunc*, and we denote $\{BGP\} \cup Sfunc$ as *SPARQL Algebra Operations (SOp).* The subset $\{Distinct, OrderBy, Limit, OffSet\} \subset Sfunc$ is called *Solution Modifiers (SMod).*

Overall we refer to $QF \cup SOp$ as *SPARQL basic operations*

Solution Mapping Multi-Sets From a technical viewpoint solution mapping multisets are unordered lists of solution mappings which may contain duplicates. For our following discussion we adopt the mathematical definition from [29]:

Definition 1. *A* solution mapping multi-set *is a tuple* $(S, card)$ *where* S *is a set of solution mappings and card is a function which annotates every* $x \in S$ *with a positive integer.*

The cardinality *of* $(S, card)$*, denoted as* $|(S, card)|$*, is the summation of* $card(x)$ *for all* $x \in S$*.*

SPARQL Query Graph Model

The *SPARQL Query Graph Model (SQGM)* [17] is a graph model for SPARQL queries resp. SPARQL Abstract Queries adopted from the *Query Graph Models* for SQL [28]. A SQGM is a planar rooted directed graph consisting of *operator* as nodes and *data flows* as edges. Each operator represents an occurrence of an SPARQL algebra operation in the corresponding SPARQL Abstract Query while each data flow connects an operator A to an operator B, iff B uses A's result as input in the corresponding SPARQL Abstract Query. Furthermore operators are divided into types each of them having a one-to-one correspondence to a SPARQL basic operation, except for BGPs in combination with Filters and also for Solution Modifiers. For the first constellation SQGMs use an operator type *Graph Pattern Operators (GPO)* which corresponds to a BGP with an optional Filter operation. For Solution Modifiers SQGMs provide an general operator type, *Solution Modifier Operators*, which can represent a concatenation of solution modifiers.

In the following we will base our cost estimation for SPARQL queries and query plans on their corresponding SQGMs. While as shown above queries can be translated directly into SQGMs query plans contain in general additional physical information for each operator. Thus we consider SQGM as abstraction of query plans in the sense that all physical information has been masked out.

Constraints for BGPs resp. Graph Pattern Operators Actually there are two different SPARQL basic operations which describe joins between solution mapping multi-sets: The *Join* operator and BGPs containing more than one triple pattern. By limiting joins between solution mapping multi-sets to the actual *Join* operation of the SPARQL Algebra we unify these to one form to ease the definition of cost and cardinality functions. Motivated by this consideration we restrict BGPs (resp. Graph Pattern Operators) to the subset of BGPs which only contains exactly one single triple pattern. As shown in the extension to [26] it is semantically equivalent to simulate BGPs embodying an arbitrary set of triple pattern by defining a BGP for each triple pattern and after that to join these.

Constrains for Solution Modifier Operators As mentioned above a Solution Modifier Operator can express a application of several Solution Modifiers. In the following we only accept Solution Modifier Operators which represent exactly one Solution Modifier. A sequence of Solution Modifier appliances $m_1, \ldots, m_n$ in a SPARQL query can also be represented in a SQGM as a path of Solution Modifiers $M_1 \rightarrow \ldots \rightarrow M_n$ where M_i solely represents m_i. This limitation is motivated by the need of a one-to-one correspondence for the sake of exchangeability of cost and cardinality functions for corresponding SPARQL basic operations and SQGMs operator types. The exchangeability of functions significantly assists the comprehensibility of the cost computations based on SQGMs.

4.2.2 Cost and Cardinality Functions

In this section we will describe the structure of the cost model for SQGMs and provide a justification to base our cost estimation on the complexities of best-practice-algorithms for relational algebra operations related to the SPARQL basic operations.

Overall the cost model for SQGMs banks on a recursive cost function relying in turn on a recursive cardinality function. Furthermore both cost and cardinality function themselves are based on functions estimating physical cost and solution cardinalities for each SPARQL basic operation.

At first we introduce a method to assign cost and cardinality functions to each SPARQL basic operation also depending on the used implementation algorithms. After that we show how we can recursively compute costs for SQGMs based on our earlier results. Finally we justify that we can reduce all functions in $Sfunc$ to their corresponding relational algebra operation in linear time to the input sizes.

Cost and Cardinality Functions for SPARQL Basic Operations

For each SPARQL basic operations we assign a *cardinality*, *CPU-* and *IO-cost function.* Together these costs represent the *physical parameters* and will be acquired by the distribution manager component through the network analysis capabilities of its underlying P2P network. The cardinality function estimates the cardinality of the result, the CPU-function the number of instructions and the IO-cost function the number of hard disk (or page) accesses. Each of these functions depends on the cardinalities of the input solution mapping multi-sets, the used algorithm and real number parameters expressing physical characteristics. If we want to compute these functions for a triple pattern we use the pattern itself as indication for the cardinality for its solution mapping multi-set. These is motivated by the fact, that either we can retrieve the cardinality directly by sending the triple pattern to the RDF-Repositories or estimate the cardinality by a selectivity method with the triple pattern as input.

In Definition 2 we define a summarization of the function assignment above, denoted as configuration. Hereby $TriplePattern$ stands for the set of all RDF Triple Pattern,$Graph$ for the set of all RDF-Graphs, $Algorithm$ for the set of all algorithms implementing SPARQL basic operations and $PhysPars$ for a mapping of identifiers to reals.

Definition 2. *A* configuration $\mathcal{C}$ *is a function assigning each SPARQL basic operation op a tuple* $\mathcal{C}(op) = (f_{CPU,op},\ f_{IO,op}, f_{card,op})$ *where*

if $op = TriplePattern$ *then* $f_{\phi,op} : (TriplePattern \times Graph \times Algorithms \times PhysPars) \rightarrow \mathbb{R}_+,\ \phi \in \{CPU, IO, card\}$,

else $f_{\phi,op} : (\mathbb{R}_+^{k_{op}} \times Algorithm \times PhysPars) \rightarrow \mathbb{R}_+,\ \phi \in \{CPU, IO, card\}$.

We call $f_{IO,op}$ *resp.* $f_{CPU,op}$ *the* IO *resp.* CPU cost function *for op and* $f_{card,op}$ *the* cardinality function *for op.*

The IO- and CPU-costs of a SPARQL basic operation can be estimated accurately by the usage of an elaborated time and space complexity function for the implementation algorithm, given that the estimation errors for the cardinalities of the input solution mapping multi-sets and the physical parameters are small.

If no additional information is available, a naive way to compute the cardinalities of the results for a SPARQL basic operation is the usage of the result cardinalities provided by the operator definitions in [29]. For instance, we could use the possible maximum of the result size. Hence such a procedure, which solely relies on the input sizes of an operator, causes generally a large estimation error. If available, statistical values for the result sizes of Triple Patterns, BGPs or more complex SPARQL Algebra Expressions are a highly recommended alternative.

Cost and Cardinality Functions for SQGM

In this section we recursively determine the execution costs and result cardinalities of a query graph by beginning the recursion at its root moving to the nodes with in-degree 0. Relevant physical cost factors like costs for swapping, hashing, computation, etc. are given in a function, *physical parameter assignement*, and thus can be respected in the cost estimation. For the delay by transmission through the network we use a very simplistic approach by assigning a real number weight for each operator *op* which indicates the net distance between the host computing *op* and the host computing *op*'s upstream operator. The actual net costs for *op* is the product ¡size of transmitted data¿ ∗ ¡distance weight¿. To determine the overall net costs for a SQGM we add up these products for each operator.

First of all we emphasize in Definition 3 the exchangebility of cost and cardinality functions for SPARQL basic operations and SQGM operator types presuming the restrictions stated in Section **??** and 4.2.1. Consequently we introduce the assignment functions for implementing algorithms, physical parameters and net-distance weights for operators. Finally we define the recursive cardinality and cost functions for (partial) SQGMs (Definition 7 and 8).

Definition 3. *Given a SQGM $(OP, DF, r, dflt, NG)$, a configuration $\mathcal{C}$ and a SQGM operator type t which represents a SPARQL basic operation op with $\mathcal{C}(op) = (f_{CPU,op}, f_{IO,op}, f_{card,op})$. We call $f_{IO,op}$ resp. $f_{CPU,op}$ the* IO *resp.* CPU cost function *for t and $f_{card,op}$ the* cardinality function *for t. Analogously we use for t the references $f_{IO,t} = f_{IO,op}$, $f_{CPU,t} = f_{CPU,op}$ and $f_{card,t} = f_{card,op}$.*

Definition 4. *For a SQGM $(OP, DF, r, dflt, NG)$ an* algorithm assignment *is a function which assigns each $x \in OP$ an algorithm.*

Definition 5. *For a SQGM $(OP, DF, r, dflt, NG)$ an* physical parameter assignment *is a function which assigns each $x \in OP$ a set of physical parameters.*

Definition 6. *For a SQGM $(OP, DF, r, dflt, NG)$ an* net-distance weight assignment *is a function which assigns each $x \in OP$ a positive real number.*

Definition 7. *For a SQGM $(OP, DF, r, dflt, NG)$, a configuration $\mathcal{C}$ and a node $x \in OP$ possessing the child nodes $c_1, ..., c_n$ the recursive function $card : OP \rightarrow \mathbb{R}_+$ called* cardinality function *is defined as follows:*

if $x \in GPO$ then $card(x) = f_{card,GPO}(tp, G)$ where tp is the triple pattern embodied in x and G is the input graph of x.

else $card(x) = f_{card,x.type}(card(c_1), \ldots, card(c_n))$ where $x.type$ denotes the operator type of x.

Definition 8. *For a SQGM $G = (OP, DF, r, dflt, NG)$, $\phi \in \{IO, CPU, NET\}$, a configuration $\mathcal{C}$, an algorithm assignment $\mathcal{A}$, a physical parameter assignment $\mathcal{P}$, a net-distance weight assignment $\mathcal{D}$ and a node $x \in OP$ the recursive function $costs_\phi : OP \rightarrow \mathbb{R}_+$ called* $\boldsymbol{\phi}$-cost function *is defined as follows:*

For $\phi \in \{IO, CPU\}$:

if $x \in GPO$ then $costs_\phi(x) = f_{\phi,GPO}(tp, G, \mathcal{A}(x), \mathcal{P}(x))$ where tp is the triple pattern embodied in x and G is the input graph of x.

else $$costs_\phi(x) = f_{\phi,x.type}(card(c_1), \ldots, card(c_n), \mathcal{A}(x), \mathcal{P}(x)) + \sum_{i \in \{1,\ldots,n\}} costs_\phi(c_i)$$

where $x.type$ denotes the operator type, $c_1, ..., c_n$ the input nodes of x.

For $\phi = NET$:

$$cost_{NET}(x) = \sum_{i \in \{1,\ldots,n\}} (\mathcal{D}(c_i) \cdot card(c_i) + costs_{NET}(c_i))$$

If x is a Select-, Describe-, Construct-, or Ask-Result Operator we refer to $costs_\phi(x)$ also as the $\boldsymbol{\phi}$-cost function for $\boldsymbol{G}$.

Cost Estimation Oriented on RDBMS Complexities

Based on [9] we will now prove that the choice of cost functions for SPARQL basic operations oriented on the complexities of best-practice-algorithms for relational algebra operations is sound. This is motivated by the fact by the strong similarities between SPARQL basic operations and the relational algebra. To support this claim we will prove that almost all SPARQL basic operations are reducible to operations of the relation algebra for RDBMS in linear time while the cardinalities of the input values are preserved.

We have to clarify that we base the following discussion on the named version of the relational algebra, as described in [1] in chapter 5.1. We define the cardinality for a database relation with duplicates as follows.

Definition 9. *A* RDB relation with duplicates *is a tuple* $(r[S], card)$ *where* $r[S]$ *is a relation for DB-schema* S *and* $card(x)$ *is a function which annotates every* $x \in r[S]$ *with a positive integer.*

The cardinality *of* $(r[S], card)$*, denoted as* $|(r[S], card)|$*, is the summation of* $card(x)$ *for all* $x \in r[S]$*.*

For the proof of Theorem 4.2.1 we introduce in advance a simple reduction algorithm to transform a solution mapping multi-set $(S, card)$ to an equivalent DB-relation with duplicates $(r[D], card')$: The union of the variables in the domains of the solutions mappings in S is schema (set of attributes) D for r (see [1]). Thus the number of domains in D is the overall number of different variables occurring in the the mappings in S. Furthermore each mapping $x \in S$ is represented by a tuple x' in r with $card'(x') := card(x)$ where x' has only values for the attributes which coincide with the domain of x (formally: attributes r who don't represent a variable in x have a globally defined *null* value, which belongs to no domain).

In the following we use the denotations: $trans(x) := x'$, $trans(S) := r$, $trans(card) := card'$, $trans((S, card)) := (r, card')$

For this algorithm we imply (without explicit proof):

1. The number of domains in D is the overall number of different variables occurring in the the mappings in S
2. $x \in S$ iff $trans(x) \in trans(S)$
3. The algorithm's time and space complexity lies in $O(|(S, card)|)$.

Theorem 4.2.1. *For two multi-sets of solution mappings* S_1 *and* S_2 *the following statements hold:*

a) If $Join(S_1, S_2) = (S, card)$ *and* $RA\text{-}Join(trans(S_1, S_2)) = (r[D], card')$ *then:*

- $x \in S$ *iff* $trans(x) \in r[D]$
- $card(x) = card'(trans(x))$ *for each* $x \in S$

b) If $Union(S_1, S_2) = (S, card)$ *and* $RA\text{-}Union(trans(S_1, S_2)) = (r[D], card')$ *then:*

- $x \in S$ *iff* $trans(x) \in r[D]$
- $card(x) = card'(trans(x))$ *for each* $x \in S$

c) If $Diff(S_1, S_2) = (S, card)$ and $RA\text{-}Diff(trans(S_1, S_2)) = (r[D], card')$ then:

- *$x \in S$ iff $trans(x) \in r[D]$*
- *$card(x) = card'(trans(x))$ for each $x \in S$ f*

d) If $LeftJoin(S_1, S_2) = (S, card)$ and $RA\text{-}LeftJoin(trans(S_1, S_2)) = (r[D], card')$ then:

- *$x \in S$ iff $trans(x) \in r[D]$*
- *$card(x) = card'(trans(x))$ for each $x \in S$*

Proof. We reduce each of the SPARQL algebra operation a) - d) (i.e. the operations which may occur in graph pattern) to an relational algebra term while preserving the input sizes.

A join of two solution mapping multi-sets (S_1, f_1) and (S_2, f_2) (in the following denoted as $Join((S_1, f_1), (S_2, f_2))$ will be reduced to a (natural) join of the representing relations r_1 for r_2 and r_2 for S_2 (in the following denoted as $\bowtie (r_1, r_2)$ extended by a duplicate counter *dups* for each embodied tuple. From the Definition of $\bowtie$ ([1] p.58, top) and the definition above of the generation of r_1 and r_2 from S_1 and S_2 follows that a solution mapping x is element of $Join((S_1, f_1), (S_2, f_2)$ iff there exists a tuple x' which has the same values for the attributes as x for its respectively variables.

In an analogous way this can be shown for $Union$ and $Diff$. $LeftJoin$ is defined as a SPARQL algebra term which combines $Join$, $Union$ and $Diff$, thus $Leftjoin$ can be reduced accordingly. □

For most of the SPARQL basic operations we can determine the complexity in the same way as in the proof for Theorem 4.2.1. Few are slightly different, e.g. *Filter* allowing regular expression in SPARQL which occasionally can not be directly translated into the selection operation δ as pointed out in [9]. For these exceptions we will estimate their complexity based on their algorithmic implementations with orientation on similar operations in the relational algebra as much as possible.

After the consideration above we are enabled to define a configuration oriented on the complexities for relational algebra operations for the example presented in Section 7.1. We use the complexity of a relational algebra operation for the cost estimation of the related SPARQL algebra operation under the assumption that a SPARQL query processor is approximately as efficient as RDBMS for the corresponding query. We will list the complexities for the SPARQL basic operations and the used algorithm. For operations which aren't reducible to RA operation in the way described in Theorem 4.2.1, we give a short description about a possible implementation based on similar RA operation (see Table 7.2).

4.2.3 Future Optimization Techniques

Having defined our cost model, we consider the other aspects of query optimisation which together with the cost model can be used to realize a distributed and adaptive semantic querying support. Here we can give at this stage some first suggestions, and will realize this in the subsequent task.

Query Decomposition, Distribution and Answer Reconstruction

In compliance to Ozcan et al. [23, 24] a SPARQL Query given to a kernel will be partitioned into sub queries which are subsequently distributed to the kernels in the system. From the answers of the sub queries the answer to the complete query can be constructed.

A query can not only be deconstructed at data flows between operators but also a single SPARQL basic operation can be decomposed. This is for instance beneficial, if a *Join* of very large solution mapping multi-sets should be evaluated, for which no kernel in the system offers enough resources for a acceptable processing time and furthermore could not be simplified by a transformation rule into a simpler query expression.

The distribution of sub queries among kernel is orientated on the costs of the sub queries and the available resources of the kernels. Admittedly the second demand doesn't scale for a Internet-sized distributed system, thus we propose as an alternative that all queries are partitioned into sub queries with approximately the same evaluation costs and are uniformly distributed among all kernels in the system.

Since generally matchings for distinct triple pattern or graph pattern are only available on a real subset of kernels or even on only one kernel sub queries are primarily only sent to kernels which hold the necessary informations.

Query Rewriting

The Query Rewriting component comprehends transformations rules which transform the query on a logical level to improve its evaluation. Thereby we can adopt transformation rules from Pirahesh et al. [28] for optimization for local execution extended with rules for distributed data sources [25].

Selection and Join Ordering

Heuristics for a cost-efficient evaluation order of data selections, i.e. triple patterns, and *Join* operations are adopted from the databases community as pointed out by [32]. From the choice of order a priorization of sub query processing jobs can be derived.

Source Indexing

In TripCom we index data sources with the "Distributed Hash Tables"-approach embodied in the Distribution Manager (see D2.4). As an adequate alternative also *Source Index Hierarchies* [32] will be considered, which basically detects upfront computed (multiple-)join results over the path the (multiple-)join describes in the RDF-schema.

Structural Query Properties

We investigate potential benefits of structural property (type) recognition for query plans as additional tool for several optimization duties. Thinkable areas of applications are query rewriting, query decomposition, cost estimation and source indexing.

To determine structural properties we rely on the fact, that SQGMs are actually rooted directed graphs, which can be amongst others described by a modification of unranked directed tree grammars, the *Regular Rooted Graph Grammars (R2G2)* [5, 6]. Hence we can describe with a concrete R2G2 a subset of all SQGMs which can be comprehended as a “type”. This is analogous to the relation between XML-documents and XML-schemas.

Intra-Query Adaptivity

At the beginning we already emphasized that the query optimizer should act in a intra-query-adaptive fashion [10]. The overall query evaluation and optimization is subdivided in several phases wereas in each phase a part of the query is optimized and executed. Thus all above-mentioned optimization techniques are applied multiple times during a complete processing of a query. In general this benefits the precision of the optimization procedure since intermediate results can be used as indication for decision making. For instance the exponential growth of the cost estimation for mutli-way joins, investigated in [18], can be softened by this measure.

5 Proposal for Inconsistent Reasoning Handling

This section proposes an approach to handle inconsistent reasoning in an open, web-based environment such as Triple Space. We focus on approaches to handle reasoning on inconsistent ontologies using the PION framework.

5.1 Handling Inconsistencies in Reasoning

There are different possible ways to deal with or avoid inconsistency that occurs in ontologies and instance data, listed below:

- During the querying process if an inconsistency is detected, stop the querying process, correct the data and schema, and execute the query again.
- Check for any inconsistencies that may arise during adding, updating or deleting statements in the ontology; or migrating the ontology from one source to another.
- Lastly, avoid the inconsistency while querying, and to apply non-standard reasoning methods to obtain meaningful answers.

The first two methods are normally suitable when we are dealing with a single or limited set of ontologies with a fixed number of users. These methods are normally used to improve an ontology in an interactive manner with its users, and may also require human intervention. These kinds of methods are useful to apply in a closed environment where it is easier to detect the fault (i.e. inconsistency), to stop, repair that and continue the process.

5.1.1 Inconsistencies in Reasoning in Triple Space

Triple Space has been envisioned as a Web for machines [13]. It is an open environment where the number of users is not known in advance. In most cases the ontologies and instance data are generated based on multi-authorship of different user (that most probably do not know each other). Moreover, there are different RDF data sources that can be added to the Triple Space dynamically which may bring several new updates from unknown sources. Therefore, the third approach (i.e. to avoid inconsistency by selecting a subset of ontologies, and apply non-standard based reasoning technique to obtain meaningful answers [34])

is more suitable for the Web like scenario in Triple Space. The inconsistent ontologies result in meaningless answers, i.e. which are logically invalid. Therefore, it is obvious that our main goal for the output of the inconsistency reasoner is to get meaningful answers. A meaningful answer as described in [36] as self-consistent as well as sound. Self-consistent answer will be the one that holds for ’A’ and does not hold for ’ A’, where ’A and A = null’. Soundness of an answer can be determined if it remains true for its own answers. A solution for detecting inconsistencies in the ontologies have been introduced in the Query PreProcessor in order to detect and avoid the inconsistencies that may arise in the ontologies published and altered by different users over the Triple Space. Different non-standard reasoning techniques have been made in order to detect the inconsistencies and to extract maximum possible meaningful answers. For this purpose, we use PION framework as a basis of our inconsistent reasoning solution.

The PION framework has been evaluated by investigating its different selection functions and testing them on large scale realistic ontologies. The first selection function that has been chosen is based on finding the syntactic connections between axioms in order to decide which axioms are relevant to a query. This relevance requirement is decreased gradually during reasoning in order to obtain a increasing subset of the axioms until it has selected a subset which is small enough to avoid the inconsistency, but large enough to answer the query. There is a second selection function that takes into account that it is dealing with ontologies and uses the concept hierarchy for the selection of related axioms. The main idea behind the design of PION framework is, given a selection function, a consistent subtheory from an inconsistent ontology is selected. Then a standard reasoning on the selected subtheory is applied to find meaningful answers. If a satisfying answer cannot be found, the relevance degree of the selection function is made less restrictive thereby extending the consistent subtheory for further reasoning. Authors identify three major properties that the inconsistency reasoner should have, i.e.

- Soundness: the argument is valid and all of its premises hold true.
- Self-consistency: the answers should be consistent to itself.
- Meaningfulness: if the the answer given is self-consistent and sound.

The inconsistency reasoner uses the selection functions to determine which consistent subsets of an inconsistent ontology can be considered during the process of reasoning. There are two major selection functions that have been considered by authors [34] in their approach, which are listed as follows:

- Selection function: if the answer of a query on the ontology (or formula set) at a step (K>0) is subset of the ontology.
- Monotonic section functions: if the subsets selected by the function monotonically increase or decrease.

In order to rate the correctness of the answers, authors use the following four states proposed by Belnaps [4]:

- Over-determined: if the result falls in the range of the query and also falls outside the range of the query (which is realistically not possible).
- Accepted: if the results falls in the range of the query and doesn't fall outside the range of the query.
- Rejected: if the results do not fall in the range of the query and only falls outside the range of the query.
- Undetermined: if the results do not fall in the range of the query and also does not fall outside the range of the query.

POIN framework has further notions to compare the quality of results. For this, authors use the notion of an intutively correct answer. Authors use following notions to capture the differences between an answer by an inconsistency reasoner and the intuitive answer (i.e. the answer that was supposed to be):

- Intended Answer: if it is exactly equal to the intuitive answer.
- Counter-intuitive Answer: if the actual answer is exactly opposite to the intended answer.
- Cautious Answer: The intuitive answer is ’accepted’ or ’rejected’, but the actual answer is ’undetermined’.
- Reckless Answer: the actual answer is ’accepted’ or ’rejected’, but the intuitive answer is ’undetermined’.

5.2 Selection Functions of Inconsistency Reasoner

An inconsistency reasoner uses a selection function to determine which consistent subsets of an inconsistent ontology should be considered in its reasoning process. The selection function can either be based on a syntactic approach, or based on semantic relevance. The RIO reasoner uses a monotonically increasing / decreasing selection function by using a linear extension strategy and a linear reduction strategy respectively. Using a linear extension strategy satisfies the following properties like answers are never over-determined or may be undetermined or always sound or always meaningful or always locally complete or may not be maximal or always locally sound.

5.3 Using PION in TripCom

Triple Space has been envisioned as a Web for machines which is characterized by scalability, distribution and multi-authorship of data (i.e. RDF triples) in it. It brings the possibility of inconsistency in ontologies and data when multiple data sources in Triple Space are integrated dynamically and multiple users globally publish, access and update the information. There can be the possibility that new logical statements, rules or axioms can be introduced by different users which may be correct when considered individually, however, may cause confusion by giving an illogical meaning when considered together. Addition of new statements about an existing ontology requires careful attention that the new statements remain consistent with the rules that already exists in the space. The inconsistency can occur in many different ways, as described in [34]. Ambiguous representations of rules can occur if new statements are added to an ontology and the new statements are not consistent with old rules. Inconsistency can also be caused by using terminologies that have dual or multiple meanings. Migrating ontologies from one data source to other data source can also cause inconsistency. Similarly creating an ontology from multiple sources can also cause inconsistencies.

There are two major steps involved while processing the inconsistency. As a first step, the inconsistency reasoner detects for any possible inconsistency that may arise. The result set is analyzed for detecting any possible inconsistency. The query is analyzed against the ontology (in this case RDF schema). Answer of the query is ranked as either over-determined, accepted, rejected or undetermined based on Belnaps four valued logic [4]. If the result is obtained as ”accepted”, it is deduced that the result is fine and no inconsistency reasoning is required. If result is obtained as over-determined, inconsistent reasoning is performed. Otherwise, result is rejected and not processed further.

In the next step, the inconsistency reasoner tries to determine which subsets of the inconsistent ontology should be considered in the reasoning process in order to find correct results. A monotonic selection function is applied that checks for a subset until it finds a consistent one. The Inconsistency Reasoner uses linear extension to find out a minimal consistent subset. The answer is checked if it is locally sound and complete for a consistent subset of the ontology. This process is repeated until it finds both a consistent subset of the ontology and a valid result against it.

6 Prototyping the Query PreProcessor

This chapter describes our implementation plans and related details for the two major components of the Query PreProcessor, i.e. Query Optimizer and Inconsistency Reasoner.

6.1 Query Optimization

The Query Optimizer (see Figure ??) supports the query processor embodied in the TS Adapter component of each kernel with optimization techniques for adaptive distributed query processing. It provides modules for cost estimation, query rewriting, join/selection ordering and query decomposition while taking the distributed processing aspect into account. The purpose and functionality of each of these modules is comprehensively explained in Section 3.3 and 4.

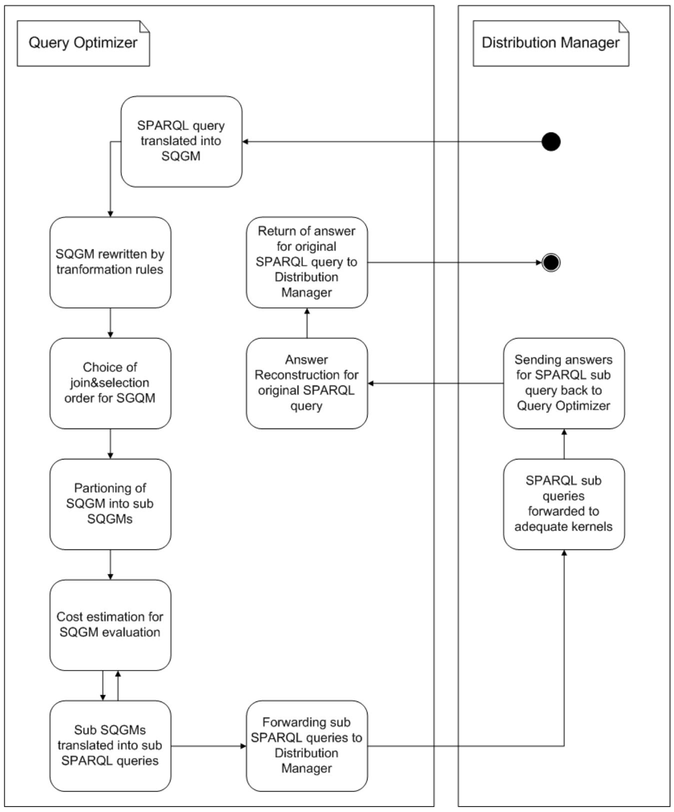
Query Optimizer
Distribution Manager
SPARQL query translated into SQGM
SQGM rewritten by tranformation rules
Return of answer for original SPARQL query to Distribution Manager
Choice of join&selection order for SGQM
Answer Reconstruction for original SPARQL query
Sending answers for SPARQL sub query back to Query Optimizer
Partioning of SQGM into sub SQGMs
SPARQL sub queries forwarded to adequate kernels
Cost estimation for SQGM evaluation
Sub SQGMs translated into sub SPARQL queries
Forwarding sub SPARQL queries to Distribution Manager

Figure 6.1: Activity Diagram for Query Optimizer and Distribution Manager

6.1.1 Module Interactions

Incoming queries are first transformed into a *Sparql Query Graph Model (SQGM)* as described in Section 4.2.1. After that several transformation rules performing logical optimizations can be applied on the SQGM by the query rewriting component. Consequently the module for the choice of the join and selection order decides in the rewritten SQGM the evaluation priority for triple patterns and triple patterns joins inside Graph Pattern Operators as well as the evaluation priority for *Join* operators.

After that the SQGM can be partioned by the decomposition module which takes the costs provided by the cost estimation into consideration for its decision making. The resulting sub queries are forwarded to the Distribution Manager addressed to the kernel best placed, based on the acquired physical parameters, to handle the sub query according to its calculated cost. The Distribution Manager who routes the sub queries to the local and remote kernels which can answer the queries. After a sub query is evaluated the Distribution Manager returns the sub query answer to the Query Optimizer where the overall answer for the original query is reconstructed by the decompositon module.

Figure 6.1 provides an overview to the interactions above.

6.1.2 Prototyping stages

On the basis of the theoretical results presented in this deliverable (Section 3.3 and 4) we will be able to turn to the concrete implementation of distributed query optimization in the next task. In a first stage we will implement the cost estimation module as theoretically described in Section 4.2. After this is accomplished the query decomposition module and necessary adjustments of the distribution manager enabling the distributed querying capabilities will be realized. Finally the refinements – the module for query rewriting and the module for join and selection ordering – will be successively added.

6.2 Inconsistent Reasoning

Inconsistency Reasoner is foreseen to provide support to the query processor to resolve and deduce the knowledge from any inconsistent data that may arise in the Triple Space. It will be based on RIO (a reasoner for reasoning with inconsistent ontologies). The RIO reasoner further relies on a standard DL reasoner (i.e. Racer).

It will evaluate the query with respect to the ontology based on the two selection functions in RIO. The first selection function finds out the syntactic connections between axioms in order to decide which axioms are relevant to a query. This relevance requirement is decreased gradually during reasoning in order to obtain an increasing subset of the axioms until it has selected a subset which is small enough to avoid the inconsistency, but large enough to answer the query. There is a second selection function that takes into account that it is dealing with ontologies and uses the concept hierarchy for the selection of related axioms.

The selection functions in RIO work in a way that a consistent subtheory from an inconsistent ontology is selected. Then a standard reasoning on the selected subtheory is applied to find meaningful answers. If a satisfying answer cannot be found, the relevance degree of the selection function is made less restrictive thereby extending the consistent subtheory for further reasoning.

6.2.1 Coordination Interface

The Inconsistency Reasoner waits for results from the Triple Space Adapter. As soon as the local results from Triple Space Adapter are ready, the Inconsistency Reasoner performs an inconsistency check over the results. During this process, it may also require to retrieve ontologies and schema information from the Triple Space Adapter.

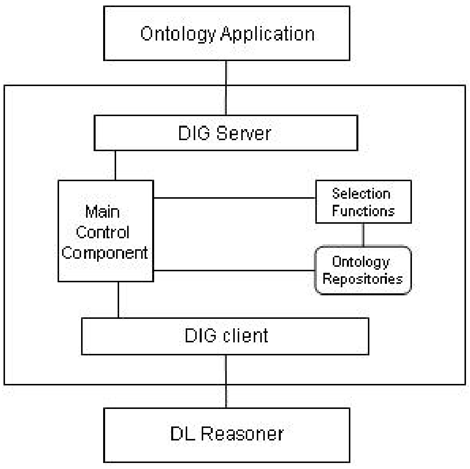

Figure 6.2: Architecture of RIO

Event Query PreProcessor gets data from Triple Space Adapter and invokes Inconsistency Reasoner to check for any inconsistencies in the ontology and the data.

Action The Inconsistency Reasoner is invoked by the Query PreProcessor upon receiving a response to a query from the TS Adapter. The control component of the RIO (which acts as the heart of the Inconsistency Reasoner) acquires the logical theories used in determination of the results, analyzes the retrieved data, using its selection functions to find and resolve any inconsistency. A standard reasoning on the selected sub-data is applied to find meaningful answers. The Inconsistency Reasoner has it's a procedure to rank the answers on the degree of meaningfulness. If a satisfying answer is found, the data is sent by the Query PreProcessor to the Distribution Manager. If a satisfying answer cannot be found, the relevance degree of the selection function is made less restrictive thereby extending the consistent subtheory for further reasoning.

6.2.2 Implementation Architecture

The RIO architecture has been designed in a way that it is independent from any DL ontology reasoner to be used as well as any applications using it. It serves as a general reasoner for inconsistent ontologies if it is accessed by its DIG interface.

The DIG Interface is a standardized XML interface to Description Logics systems developed by the DL Implementation Group [25]. It helps in keeping the description logic systems independent from the underline reasoners.

The RIO reasoner relies on a standard DL reasoner, which can either be integrated, or called as an external component. RIO will be based on those external DL reasoners. Another alternative will be to integrate the reasoning into the TS Adapter, which already provides reasoning capabilities.

DIG Server: RIO's DIG server deals with any request from other ontology applications. It supports an extended DIG interface. Namely, it supports not only the standard DIG requests, like 'tell' and 'ask', but also additional reasoning processing, like the identification or change of the selected selection functions.

The architecture of RIO can be seen as follows:

The main control component realizes the main processing, like query analysis, query PreProcessing, and the extension strategy, by calling the selection function and interacting with the ontology repositories. It acts as manager and the core component

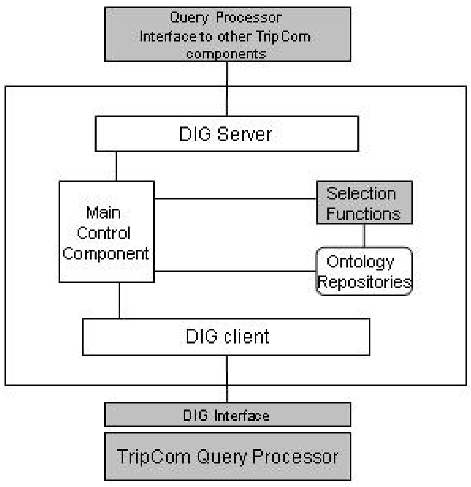

Figure 6.3: RIO inconsistency reasoning with Triple Space query processor

of the RIO reasoner. The selection function component defines the selection functions that are used in the reasoning process. RIO's DIG client calls external DL reasoners which support the DIG interface to obtain the standard DL reasoning capabilities. Ontology Repositories are used to store the ontology statements which are provided by external ontology applications.

6.2.3 RIO with Triple Space Querying Mechanism

The RIO has been implemented in Prolog. Therefore, for a first proof-of-concept prototype, the strategy has been to use the RIO system with the Triple Space query processor as an external process and to analyze the current selection functions (as well as try to propose new, if possible) used in RIO. There are following requirements that are imposed:

- The other components in Triple Space will be accessing the query processor as typical ontology applications accessing a reasoner. The query processor interface is needed to be made compatible with DIG interface to be able to use RIO via its DIG interface.
- The query processor will also have to be provided with the DIG interface so that RIO can access it for getting the data from Triple Space.
- It is not necessary to rely only on already existing selection function used by RIO. The current selection functions will be analyzed and a new selection functions might also needed to be proposed if required.

6.2.4 Activity Diagram

The Inconsistency Reasoner does not have to interact with other components of Triple Space except the query processor. The query processor will call the Inconsistency Reasoner while querying the Triple Space in order to get any inconsistency resolved if required, as shown in the activity diagram below:

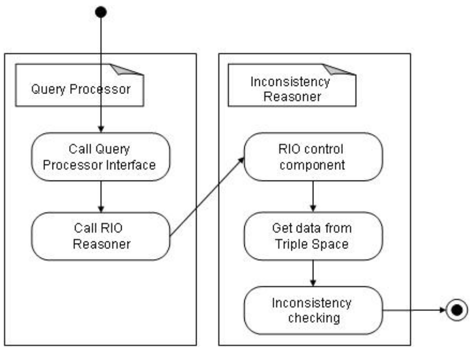

Figure 6.4: Activity diagram for interaction of Query PreProcessor and Inconsistency Reasoner

7 Evaluation results

In this chapter the benefits of the Query Optimizer and Inconsistency Reasoner are demonstrated by examples.

7.1 Results from Query Optimizer

At this point we showcase the expected behavior and results from the Query Optimizer. We provide an exemplary SPARQL query q (Listing 7.1) and demonstrate how the optimizer's key components, cost estimation and query decomposition, will established an optimized distributed evaluation of the query. The other two components, query rewriting and join/selection ordering, are left out in this example since their factual design is part of the next task. The absence of these components doesn't hurt the verification of the overall distributed processing concept since these have only a refinement character.

For this example we make the following presumptions for the kernel hosting machines:

- The RAM size equals one block of persistent memory
- One I/O access equals reading of one block in persistent memory
- The RDF triples in the RDF repositories are indexed by a B+ tree as described in [15]
- Statistics about cardinalities of solution mapping multi-sets for SPARQL Algebra Expressions are available for each RDF repository (e.g. for BGPs a schema-path-based index of the result cardinalities for n-way triple pattern joins would be possible, analogous to the source index hierarchy proposed by [32])

In Table 7.2 we provide a list of IO and CPU cost functions necessary for this example, i.e. for each occurring operator type. These functions were derived from [14, 12]. Table 7.1 explains the used symbols.

```
PREFIX ub:<http://www.lehigh.edu/.../univbench.owl#>
PREFIX rdf:<http://www.w3.org/1999/02/22−rdf−syntax−ns#>
SELECT ?n ? c
FROM <http://example.or/University0.owl>
WHERE {
    ?s rdf:type ub : GraduateStudent .
    OPTIONAL { ?s ub:takesCourse ?c . }
    ?s ub:name ?n .
}
```

Listing 7.1: SPARQL query q

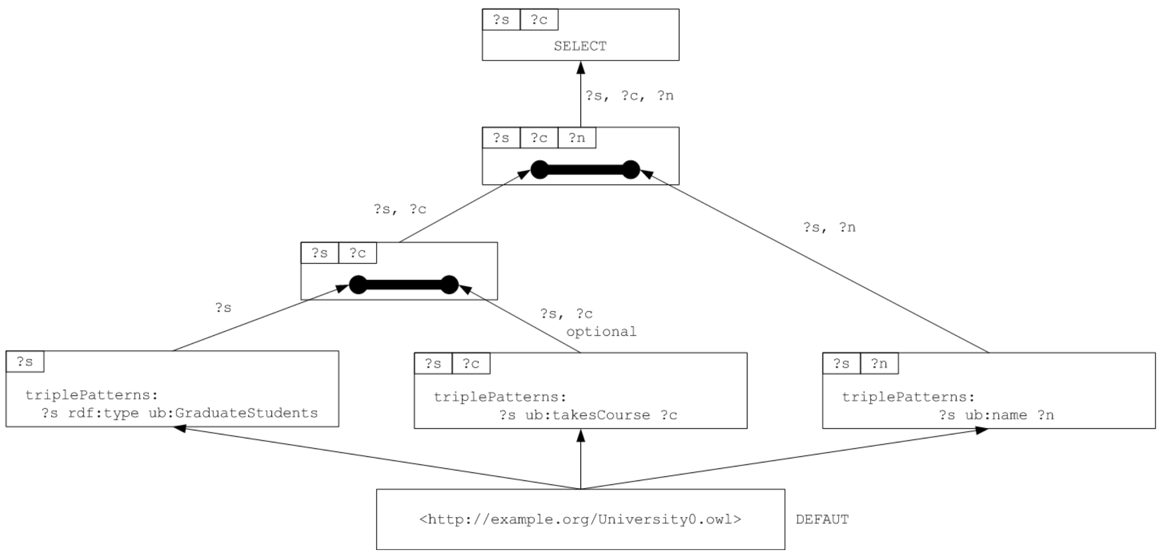

Figure 7.1: Corresponding SQGM G for SPARQL query q

n_i	cardinality of i-th input solution mapping multi-set
b_i	blocks used by i-th input solution mapping multi-set
b_{res}	blocks used by the resulting solution mapping multi-set, i.e $\lceil n_{res}/b_{kernel} \rceil$
n_{res}	cardinality of the resulting solution mapping multi-set
l_i	average number of variables in i-th input solution mapping multi-set
$c_{generic}$	generic cost factor (if no other symbol is fitting)
$c_{compare}$	number of instructions to compare two variable bindings
c_{swap}	costs for swapping a block
c_{hash}	number of instructions to hash a key
n_{space}	number of RDF triple stored in the space
b_{kernel}	block size of the kernel, i.e. RDF triples per block

Table 7.1: Symbols used in the cost and cardinality formulas

Table 7.2: Cost functions for SQGM operator types

operator type	algorithm	cost function
Select Result Operator	full table scan	$f_{CPU} = n_1 \cdot c_{generic}$ $f_{IO} = b_1 \cdot c_{generic}$
Join Operator	NestedLoop-Join	$f_{CPU} = n_1 \cdot n_2 \cdot c_{compare}$ $f_{IO} = b_1 \cdot b_2 \cdot c_{swap}$

	Hash-Join	$f_{CPU} = min(n_1, n_2) \cdot c_{comp} + max(n_1, n_2) \cdot c_{hash}$ $f_{IO} = (b_1 + b_2) \cdot c_{swap} + b_{res} \cdot c_{generic}$
	MergeSort-Join	$f_{CPU} = (n_1 \cdot log n_1 + + n_2 \cdot log n_2 + + n_1 + n_2) \cdot c_{compare}$ $f_{IO} = (b_1 \cdot log b_1 + + b_2 \cdot log b_2 + + b_1 + b_2) \cdot c_{swap}$
LeftJoin Operator	Union(HashJoin,Diff)	$f_{CPU} = f_{CPU,Hashjoin}(n_1, n_2) + + (n_1 + n_2) \cdot c_{compare} + + n_1 \cdot c_{hash}$ $f_{IO} = f_{IO,Hashjoin}(b_1, b_2) + + (b_1 + b_2) \cdot c_{swap} + + b_1 \cdot c_{generic}$
Graph Pattern Operator consisting only of a single triple pattern	B+-Tree lookup	$f_{CPU} = c_{comp} \cdot log_{b_{host}} n_{repo}$ $f_{IO} = c_{swap} \cdot (log_{b_{host}} n_{repo} + b_{res})$

Now we give a step by step walk-through of the distributed querying process:

1. Initially SPARQL query q (Listing 7.1) is transformed into a SQGM G (Figure 7.1).
2. Then the Decomposition Component determines – by calling the Cost Estimation Component – the CPU-costs and IO-costs for each operator and thereby for each sub graph in SQGM. We use the following randomly chosen configuration of physical characters for each operator in the SQGM G: $c_{generic} = c_{compare} = 10$, $c_{swap} = 200$, $c_{hash} = 20$, $n_{space} = 10^{12}$, $b_{kernel} = 100$
3. After the Decomposition Component gets back the cost estimations (Figure 7.2) it decides the partitioning of SQGM G with one of the following strategies (These strategies are intuitive choices, their sophisticated research is targeted in the next task.)

If the query optimizer is regularly notified by the Distribution Manager about available resources, the partitions can be chosen w.r.t. to their costs and these resources, i.e. a matching between costs and resources is approximated. Alternatively the Decomposition Component tries to partition SQGM G into chunks with circa the same costs. Here we use the second strategy.

The overall costs for the left and right sub-tree (Assumption: Default Graph operator in SQGM G is masked) at the upper-most *Join* operator are $3400 + 340 = 3730$ and $2500 + 250 = 2750$ if we weight CPU and IO costs with factor 1. Two remote kernels A and B (detected by the Distribution Manager) should be available with access to q's default graph. A should have net-distance 2 and B net-distance 3 to the kernel which will process the uppermost *Join* operator and the Select operator. The remote processing costs for the left subtree on kernel A would than be 3730 plus the net latency $3 \times 200 = 600$, for the right subtree on kernel B 2750 plus the net latency $2 \times 250 = 500$. Since A and B work parallel the overall cost is 4330 (if we have to wait until A is finished) for the remote processing of the both sub trees compared to a sequential local processing which is $3730 + 2750 = 6480$. Thus we split G at the uppermost-most *Join* operator.

4. The resulting partitions for the sub trees of the uppermost *Join* operator in G lead to the sub queries q_1 and q_2 (Listings 7.1 and 7.1), which then are forwarded by the Distribution Manager to kernel A and B for processing. The retrieved sub query answers routed back by the Distribution Manager are combined to an answer for q according to its decomposition.

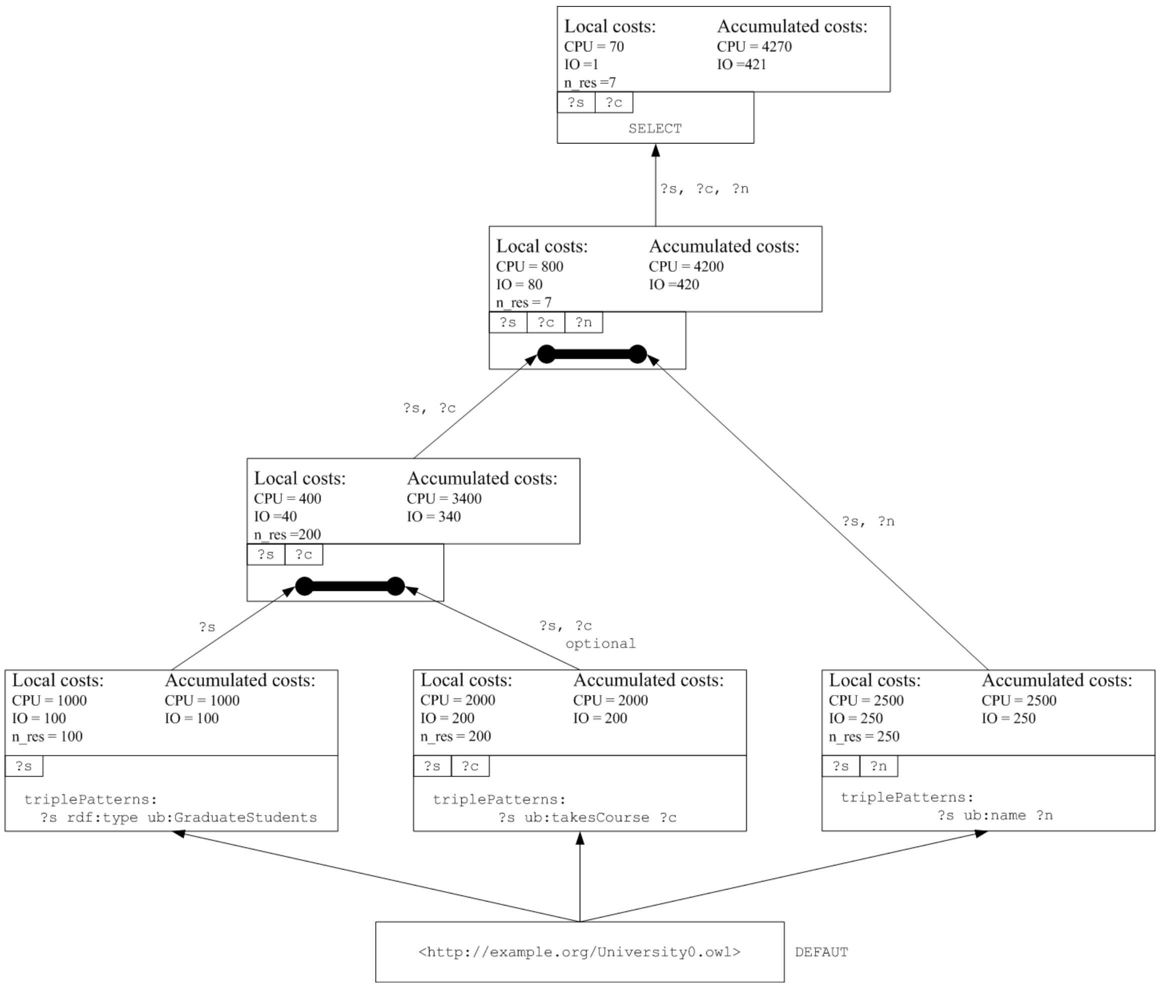

Figure 7.2: Cost annotated SQGM G

```
PREFIX ub:<http://www.lehigh.edu/.../univbench.owl#>
PREFIX rdf:<http://www.w3.org/1999/02/22−rdf−syntax−ns#>
SELECT ?s ? c
FROM <http://example.or/University0.owl>
WHERE {
    ?s rdf:type ub : GraduateStudent .
    OPTIONAL { ?s ub:takesCourse ?c . }
}
```

Listing 7.2: SPARQL query $q1$

```
PREFIX ub:<http://www.lehigh.edu/.../univbench.owl#>
PREFIX rdf:<http://www.w3.org/1999/02/22−rdf−syntax−ns#>
SELECT ?s ?n
FROM <http://example.or/University0.owl>
WHERE {
    ?s ub:name ?n .
}
```

Listing 7.3: SPARQL query $q2$

7.2 Results and Evaluation Plans for Inconsistency Reasoner

This section presents the testing and evaluation plans once the Inconsistency Reasoner for Query PreProcessor will be implemented. Our approach will be to evaluate and measure the behavior of inconsistency reasoner for the ontology that are simple in size as well as expressivity, as a first step. We will perform evaluation by creating some sample inconsistent ontologies, run a set of queries on these ontologies with and without using inconsistency reasoner, and will compare the difference. The inconsistency reasoner will be evaluated in two ways, i.e. the correctness of functionality and performance measure (time taken). There are different parameters that have been considered for the evaluation of functionality and performance of the inconsistency reasoner. These parameters are relevance based on a selection function, number of queries executed on an ontology and the number of different categorized answers obtained by running those queries on the ontologies, as well as rate of obtaining intended answers with respect to the queries executed. The further efficiency results will be obtained in terms of performance of the inconsistency reasoner.

We further present the intended results that are expected to be obtained from Inconsistency Reasoner in the Query PreProcessor. Two examples have been presented below that shows how the inconsistency in the ontologies and data can occur while multiple users are accessing Triple Space. This is predicated on the use of knowledge models which are more expressive than RDF, in which inconsistency can occur. In the first example, this lies in the use of disjoint classes (OWL). In the second example we use WSML in order to see how inconsistency can occur.

7.2.1 Example 1

In this section, we present an example to show how exactly inconsistencies can occur when using OWL and disjoint classes.

The example below illustrates the inconsistency that may occur due to ambiguous description of concepts and their relationships. The example describes some rules as follows:

- 1. Birds are animals.
- 2. Birds are flying animals.
- 3. Eagle is bird
- 4. Penguin is bird
- 5. Penguin is not flying animal.

The triples for the statements given below can be seen as follows:

```
zoo:Bird rdf:type rdfs:Class.
zoo:Animals rdf:type rdfs:Class.
zoo:FlyingAnimals rdf:type rdfs:Class.
zoo:NonFlyingAnimals rdf:type rdfs:Class.
zoo:Eagle rdf:type rdfs:Class.
zoo:Penguin rdf:type rdfs:Class.

zoo:Bird rdfs:subClassOf zoo:FlyingAnimals.
zoo:FlyingAnimals rdfs:subClassOf zoo:Animals.
zoo:NonFlyingAnimals rdfs:subClassOf zoo:Animals.
```

```
zoo:FlyingAnimals owl:disjointWith zoo:NonFlyingAnimals.

zoo:Eagle rdfs:subClassOf zoo:Bird.
zoo:Penguin rdfs:subClassOf zoo:Bird.
zoo:Penguin rdfs:subClassOf zoo:NonFlyingAnimals.
```

Listing 7.4: RDF triples

For the above mentioned RDF statements, one can conclude that penguins can fly, which is actually not true, as they are birds and birds are flying animals. However, penguins are also non-flying animals which is a disjoint class to flying animals. Hence it can generate a logically incorrect result and may cause confusions. Sample SPARQL query is given below which may give the result which may be correct for the given RDF statements but is logically inconsistent.

SPARQL query:

```
SELECT ?zoo:birds WHERE (?x rdf:type zoo:FlyingAnimals)
```

Listing 7.5: SPARQL Query

Result of the query will be:

- Eagle
- Penguin

”Penguin” is an undesired result, even though by subsumption it is correct. This finding should be omitted by the Inconsistency Reasoner based on its selection function, where it is recognized that the subclassing of Penguin with the disjoint class NonFlyingAnimal should take precedence in the evaluation of axioms.

7.2.2 Example 2

The example below illustrates the inconsistency that may occur due to usage of concept words that have multiple meaning. The example describes some rules as follows:

- 1. A married woman is a woman.
- 2. A married woman is not a divorcee.
- 3. A divorcee had a husband and has no husband.
- 4. Has husband means married
- 5. Had Husband means married

The WSML ontology for the statements given below can be seen as follows:

```

concept Woman

concept Husband

concept MarriedWoman subConceptOf Woman

concept Divorcee subConceptOf Woman

relation hasHusband(ofType MarriedWoman, ofType Husband)

relation hadHusband(ofType Divorcee, ofType Husband)


axiom a1
definedBy
  ?x memberOf Woman and ?y memberOf Husband and hasHusband(?x,?y) implies ?x memberOf MarriedWoman.

axiom a2
definedBy
  ?x memberOf Woman and ?y memberOf Husband and hadHusband(?x,?y) implies ?x memberOf MarriedWoman.


axiom marriedWomanNotDivorcee
definedBy
!− ?x memberOf MarriedWoman and ?x memberOf Divorcee.


instance lady1 memberOf MarriedWoman
instance lady2 memberOf MarriedWoman
instance lady3 memberOf MarriedWoman

instance lady4 memberOf Divorcee
instance lady5 memberOf Divorcee
```

Listing 7.6: WSML Ontology

For the above mentioned WSML ontology statements, one can conclude that divorcee is a married woman which has actually two different meanings in real world - currently married or once married? Hence it can generate a logically incorrect result and may cause confusion. A sample query is also given below which may give the result which may be correct on the basis of some ontology statements but is logically incorrect.

```

?x memberOf MarriedWoman
```

Listing 7.7: Query for the WSML Ontology

The query in listing wants to know who is a married woman. Result of the query will be:

- Divorcee
- MarriedWoman

The "divorcee" is the undesired result, which will be omitted by the Inconsistency Reasoner based on its selection function, in which axiom a2 in the listing above is rejected based on the contradiction occurring through the axiom marriedWomanNot-Divorcee and the definition of property hadHusband.

The testbed of the Inconsistency Reasoner will be developed and different examples on it as mentioned above, will be applied. The results of the Inconsistency Reasoners will be compared with the intutive answers which are actually supposed by the users.

8 Conclusion

The work in WP3 will continue via the development of the Query PreProcessor component. In this deliverable we established the basis for this by specifying its two aspects: query optimisation and inconsistency handling. Both aspects are significant for the support of distributed semantic querying in Triple Space. We defined a novel cost model for SPARQL queries which is a core part of the optimisation subcomponent, and realised a concept for inconsistency reasoning using the PION framework.

The Query PreProcessor component will be evaluated as to whether the preprocessing of the query expressions and their rewriting and decomposition into several queries will deliver better scalability than evaluating the original query as such. In addition, the integration of the PION framework to the Query PreProcessor component will be evaluated on a standalone prototype basis (external to the Triple Space Kernel development) whether it delivers the required inconsistency management as might arise in the TripCom use cases.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading, Massachusetts, 1995.

[2] L. J. B. N.
Kia Teymourian
Reto Krummenacher
Vassil Momtchev
Alessandro Ghioni
Adi Schütz. Semantic clustering and self-organization in triple space. TripCom Project Deliverable D2.4, July 2008.

[3] G. Antoniou and A. Bikakis. DR-prolog: A system for defeasible reasoning with rules and ontologies on the semantic web. *IEEE Trans. Knowl. Data Eng*, 19(2):233–245, 2007.

[4] N. Belnap. A useful four-valued logic. *In Modern Uses of Multiple-Valued Logic*, pages 8–37, 1977.

[5] S. Berger and F. Bry. Towards static type checking of web query language. In S. Brass and C. Goldberg, editors, *17th GI-Workshop on the Foundations of Databases*, pages 28–32, Wörlitz, Germany, May 2005.

[6] S. Berger, E. Coquery, W. Drabent, and A. Wilk. Descriptive typing rules for xcerpt. In F. Fages and S. Soliman, editors, *PPSWR*, volume 3703 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2005.

[7] M. Cai and M. Frank. Rdfpeers: a scalable distributed rdf repository based on a structured peer-to-peer network. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 650–657, New York, NY, USA, 2004. ACM.

[8] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43. ACM Press, 1998.

[9] R. Cyganiak. A relational algebra for SPARQL. 2005.

[10] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[11] D. L. M. L. Diego Calvanese, Giuseppe De Giacomo and R. Rosati. Inconsistency tolerance in p2p data integration: an epistemic logic approach. In *in Proc. of the 10th Int. Workshop on Database Programming Languages (DBPL 2005)*, 2005.

[12] R. Elmasri and S. B. Navathe, editors. *Fundamentals of Database Systems.* Benjamin/Cummings, second edition, 1994.

[13] D. Fensel. Triple-space computing: Semantic web services based on persistent publication of information. In *In Proceedings of the IFIP International Conference on Intelligence in Communication Systems, INTELLCOMM 2004*, Bangkok, Thailand, 2004.

[14] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[15] A. Harth and S. Decker. Optimized index structures for querying rdf from the web. In *LA-WEB '05: Proceedings of the Third Latin American Web Congress*, page 71, Washington, DC, USA, 2005. IEEE Computer Society.

[16] A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. J. B. Nixon, J. Golbeck, P. Mika, D. Maynard, G. Schreiber, and P. Cudré-Mauroux, editors, *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea*, volume 4825 of *LNCS*, pages 211–224, Berlin, Heidelberg, November 2007. Springer Verlag.

[17] O. Hartig and R. Heese. The SPARQL query graph model for query optimization. In E. Franconi, M. Kifer, and W. May, editors, *ESWC*, volume 4519 of *Lecture Notes in Computer Science*, pages 564–578. Springer, 2007.

[18] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 268–277, New York, NY, USA, 1991. ACM.

[19] L. N. A. P. D. d. F. Janne Saarela, Tommi Koivula. State of the art and triple space-specific requirements of semantic query languages. TripCom Project Deliverable D3.2, March 2007.

[20] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv*, 32(4):422–469, 2000.

[21] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *Twelfth International Conference on Very Large Data Bases*, pages 149–159, Kyoto, Japan, 25–28 Aug. 1986. Morgan Kaufmann.

[22] D. Nute. Defeasible logic. In *INAP*, pages 87–114, 2001.

[23] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, and A. Dogac. Dynamic query optimization on a distributed object management platform. In *Proc. of Fifth International Conference on Information and Knowledge Management (CIKM '96)*, Nov. 1996. Maryland, USA.

[24] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, and A. Dogac. Dynamic query optimization in multidatabases. *IEEE Data Eng. Bull*, 20(3):38–45, 1997.

[25] M. Panti, L. Spalazzi, and L. Penserini. A distributed case-based query rewriting. In *IJCAI*, pages 1005–1010, 2001.

[26] J. Perez, M. Arenas, and C. Gutierrez. The semantics and complexity of SPARQL. 2006.

[27] M.-C. R. Philippe Chatalic, Gia Hien Nguyen. Reasoning with inconsistencies in propositional peer-to-peer inference systems. In *European Conference on Artificial Intelligence (ECAI 2006)*, pages 352–356, 2006.

[28] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In M. Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2–5, 1992*, volume 21(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 39–48, pub-ACM:adr, 1992. ACM Press.

[29] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, Jan. 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[30] E. Ruckhaus, E. Ruiz, and M.-E. Vidal. Query evaluation and optimization in the semantic web, 2007.

[31] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path election in a relational database management system. *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 23–34, May 30 - June 1, 1979.

[32] H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index structures and algorithms for querying distributed RDF repositories. In S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors, *WWW*, pages 631–639. ACM, 2004.

[33] H. J. ter Horst. Combining rdf and part of owl with rules: Semantics, decidability, complexity. In *in the proceedings of International Semantic Web Conference (ISWC 2005)*, Galway, Ireland, 2005.

[34] A. t. T. P. G. Zhisheng Huang, Frank van Harmelen and C. Visser. Reasoning with inconsistent ontologies: a general framework. In *Deliverable D3.4.1 - EU-IST Integrated Project (IP) IST-2003-506826 SEKT (SEKT: Semantically Enabled Knowledge Technologies)*, 2005.

[35] F. v. H. Zhisheng Huang. Reasoning with inconsistent ontologies: Evaluation. In *Deliverable D3.4.2 - EU-IST Integrated Project (IP) IST-2003-506826 SEKT (SEKT: Semantically Enabled Knowledge Technologies)*, 2006.

[36] F. v. H. Zhisheng Huang and A. ten Teije. Reasoning with inconsistent ontologies. In *in the proceedings of International Joint Conference on Artificial Intelligence (IJCAI 2005)*, Edinburgh, Scotland, 2005.