TripCom

Triple Space Communication

FP6 – 027324

D3.4
Distributed Semantic Query Tool for Triple Space

Lyndon Nixon
Philipp Obermeier
Sebastian Dill
Janne Saarela
Pasi Tiitinen

EXECUTIVE SUMMARY

This document is targeted for software developers and project managers who wish to exploit the TripCom project results. This document describes the background for the development of the Distributed Semantic Query Tool (DSQT) and Query Preprocessor (QPP). Those who wish to start using these tools immediately should look for the user guide for DSQT.

This document lays out the state-of-the-art in the domain of semantic query tools developed to support construction of SPARQL based queries. It then lays out requirements for such a tool in the context of Triple Space computing. Finally we describe the implementation details of both DSQT and QPP.

DOCUMENT INFORMATION

IST Project Number	FP6 – 02734	**Acronym**	TripCom
Full Title	Triple Space Communication		
Project URL	http://www.tripcom.org		
Document URL			
EU Project Officer	Werner Janusch		

Deliverable	**Number**	3.4	**Title**	Distributed semantic query tool for Triple Space
Work Package	**Number**	3	**Title**	Triple Space Interaction

Date of Delivery	**Contractual**	M33	**Actual**	
Status	version 12		final □	
Nature	prototype □ report X dissemination □			
Dissemination level	public X consortium □			

Authors (Partner)	Janne Saarela, Pasi Tiitinen (Profium), Lyndon Nixon, Philipp Obermeier, Sebastian Dill (FUB)			
Resp. Author	Janne Saarela		**E-mail**	janne.saarela@profium.com
	Partner	Profium	**Phone**	+358 (0)9 855 98 000

Abstract (for dissemination)	This document describes the background and raison d'être for Distributed Semantic Query Tool and Query Preprocessor in the context of the TripCom project.
Keywords	SPARQL

PROJECT CONSORTIUM INFORMATION

Leopold Franzens University Innsbruck http://www.deri.at	LFUI	LFUI Prof. Dr. Dieter Fensel Digital Enterprise Research Institute (DERI) Innsbruck, Austria E-mail: dieter.fensel@deri.org
National University of Ireland, Galway http://www.deri.ie	NUIG	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	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	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	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	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	Profium Dr. Janne Saarela Profium OY Espoo, Finland E-mail: janne.saarela@profium.com
CEFRIEL SCRL. http://www.cefriel.it/	Cefriel	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

LIST OF ABBREVIATIONS

ANSI	American National Standards Institute
BSD	Berkeley Software Distribution
CVS	Concurrent Versioning System
DAWG	Data Access Working Group
DBMS	Database Management Systems
DSQT	Distributed Semantic Query Tool
ER	Entity Relationship
FOAF	Friend Of a Friend
GPL	GNU General Public Licence
GWT	Google Web Toolkit
HTTP	Hyper Text Transfer Protocol
iTQL	Interactive Tucana Query Language
JRDF	Java RDF
LAN	Local Area Network
LFUI	Leopold-Franzen-Universität Innsbruck
LGPL	GNU Lesser General Public Licence
N3	Notation 3
N3QL	N3 Query Language
NDM	Oracle Spatial Network Data Model
OASIS	Organization for the Advancement of Structured Information Standards
ORDI	Ontology Representation and Data Integration
OWL	Web Ontology Language
OWLIM	OWL In Memory
QPP	Query Preprocessor
RDBMS	Relational DBMS
RDF	Resource Description Framework
RDFS	RDF Schema
RDQL	RDF Data Query Language
RPC	Remote Procedure Call
SAIL	Storage And Inference Layer
SOFA	Simple Ontology Framework API
SOAP	Simple Object Access Protocol
SeRQL	Sesame RDF Query Language
SEQUEL	Structured English Query Language
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
TAP	The Alpiri Project
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WSDL	Web Service Description Language
WSMO	Web Service Modeling Language
XML	Extensible Markup Language
YARS	Yet Another RDF Store
YARSQL	YARS Query Language

1 INTRODUCTION

In this deliverable we discuss two components of the Triple Space Communication project, Distributed Semantic Query Tool (DSQT) and Query Preprocessor(QPP). DSQT is a piece of software that operates on the Triple Space architecture. It is provided for software developers who wish to make use of TripCom project results and its goal is to assist in the design time activities in developing new applications.

DSQT is designed to work across various environments by being a thin-client application where the user can navigate the Triple Space spaces and construct queries which make use of the W3C Recommendation [7] for querying Semantic Web metadata repositories such as Triple Space. The queries can also be saved for later reference or can be immediately used in application code. Naturally they can be executed and also evaluated for eventual execution cost.

This document describes the requirements set for this piece of software, the state of the art in this field, the actual implementation and finally an evaluation where the match with the requirements is verified.

This document is not an end-user guide – a separate document is provided for those who wish to exploit the tool without understanding the full background for it.

The visionary idea behind the Triple Space System is the establishment of a mechanism to publish communication data between Web Services, according to the Web paradigm of 'persistently publish and read'. In terms of the inherent distributed structure of the WWW, the search requests of Web Services primarily span over multiple databases. Hence, in the current version of the TripCom kernel, a 'rd without space' query yields a result, only if an according set of triples is located within a single space. The *Query Preprocessor* (*QPP*) improves upon this situation by providing the means of *adaptive distributed querying*. Essentially, this technique describes how to decompose a SPARQL query into subqueries, distribute the subqueries to kernels for evaluation and, eventually, reassemble the answers for the subqueries to the overall result for the initial query. Thereby all actions are executed with respect to logical properties of the query, physical characteristics of the distributed environment and database statistics. Furthermore, considering the TripCom provides no perfect recall guarantees at all for 'rd without space', we will strongly align our optimization measures in the QPP to return only a few of all solution bindings for a SPARQL query to be solved. As we are allowed to disregard high recall rates, we will pay high attention on the execution time of the queries.

DSQT makes use of QPP at both query planning and query execution time. In planning time, DSQT can ask QPP for the estimated evaluation cost of the query being constructed. In execution time, the query clicked to be executed in DSQT, will be processed by QPP for eventual rewriting and decomposition before sent on for actual evaluation. Both DQST and QPP are integrated with the TripCom architecture as illustrated in Figure 1.1.

The QPP is integrated as subcomponent in the Distribution Manager which accepts ‘rd without space’ and ‘getDistributedQueryingCosts’ requests sent through the TS API by TS clients and the DSQT, respectively. Moreover, acting as a TS Client the

QPP transparently forwards subqueries and metadata requests to the local and remote kernels by ‘rd with space’ requests.

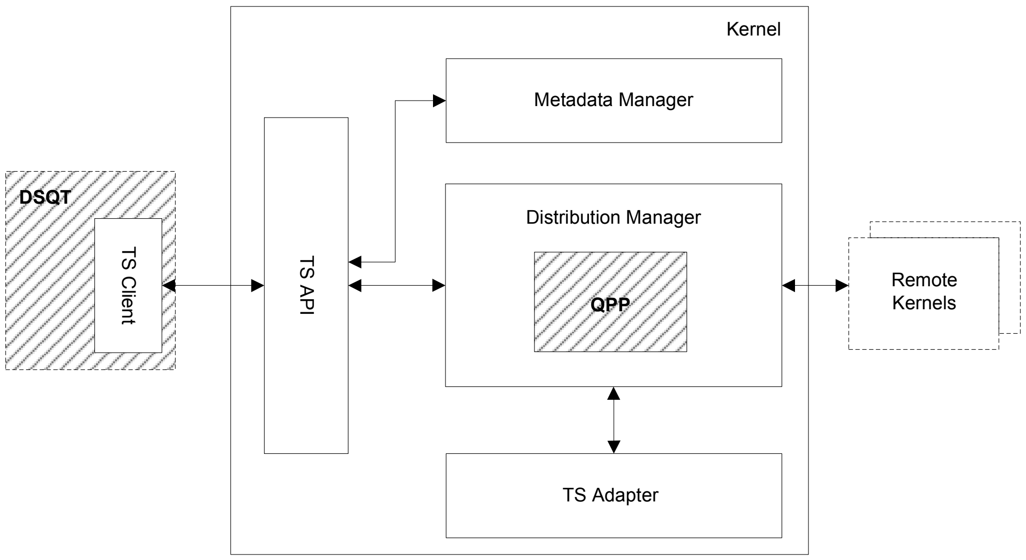


Figure 1.1: Relationship of DQST and QPP in the TripCom System Architecture

In this deliverable we will introduce a refined concept of the QPP. First, we analyze the necessary requirements from the TripCom System to our distributed query procedure. After that, we continue with an overview of already explored concepts in the field of distributed conjunctive RDF querying. Subsequently, we describe the conceptual architecture of the QPP with an emphasis of the elementary stages -- *parsing*, *logical optimization*, *decomposition*, *physical optimization* and *execution*. Eventually, we present our evaluation results and emphasize some significant aspects we encountered during the technical implementation and integration of the QPP into the existing TripCom kernel.

2 TRIPLE SPACE REQUIREMENTS

In this chapter the requirements for the Semantic Query Tool and the Query Pre-processor are described. The words "must", "required", "shall", "should", and "may" are to be interpreted in a manner similar to that described in IETF RFC 2119 [1].

2.1 Requirements for Semantic Query Tool

We have identified the following requirements for the role of a software developer who is in the process of writing a new application that works with Triple Space.

- Tool must support activities of the Triple Space developers by allowing its users to write valid SPARQL[7] queries. This requirement makes the learning and writing of SPARQL language more efficient.
- Tool must allow its users to evaluate SPARQL queries against one or many triple space(s). This requirement allows users to use the tool in conjunction with different triple spaces which might manage different types of metadata in one seamless user interface.
- Tool must allow its users to visualize query results in an environment where the query expression can be further edited and evaluated. This parallel environment allows the user to fine-tune the query given the immediate results from its execution.
- The constructed query expression and resulting query results must both be storable in standard format in local hard drive for later re-use. This requirement allows efficient re-use of the constructed queries in DSQT and in other SPARQL enabled applications.
- The query execution cost may be explained to the user prior to its execution. This requirement allows the user to understand the eventual execution cost of the query being constructed and (s)he can then make changes to the query to make it as efficient as possible. In addition, this requirement is standard practice in SQL based environments and the tool must meet with users who expect this functionality to be in place.
- The query evaluation completeness may be explained to the user after the query execution. This requirement allows user to understand if the query result is complete or not and make informed decisions about the query result.
- Tool should show metadata about the Triple Spaces being queried. This includes information such as number of kernels, available spaces and relationships between spaces. This requirement allows the user to understand better the data environment where the query is executed instead of having to operate against a black box without any understanding of its internals.

2.2 Requirements for Query Preprocessor

As a major goal, a seamless integration of the QPP into the Triple Space system has to be accomplished. Subsequently, we will enumerate the major steps that have to be taken for this matter.

The TripCom kernel encapsulates SPARQL queries in rd operations, consequently, this convention must be considered by the QPP for each SPARQL query received or forwarded. Moreover, since the QPP, by concept, does not target a concrete kernel as data source, only 'rd without space' queries are handled by the QPP.

In the previous approaches for querying distributed SPARQL endpoints, introduced in Section 3.2, a catalogue of service descriptions of all endpoints in the distribute environment were kept locally at the machine that run the query engine and initially receives the query. However, this does not comply with the strictly distributed nature inherent to the Triple System architecture; no kernel can hold and maintain a global database of the service description for all kernels in the Triple Space System. Furthermore, each TripCom kernel must be equivalent in terms of functionality; an appointment of a distinct kernel to run the distributed query engine contradicts this prerequisite. Consequently, the global catalogue of service descriptions has to be stored in a distributed fashion across the kernels and must be accessible for each kernel. Furthermore, we have to consider a method how a kernel can measure and log its own metadata. These will provide a basis for query decomposition and physical optimization during query processing. For both these matters, a solution will be elaborated in Section 4.1.3.

Another issue to be considered is the distribution of subqueries, created during distributed query processing by partitioning a client's query, and the retrieval of the subqueries' results. For these purposes we will reuse the Distribution Manager (DM) -- the component of the Triple Space System providing the routing means for query distribution and retrieval of results. More precisely, the QPP will be integrated as a new subcomponent of the DM. As such it can directly take advantage of the DM's facilities. This issue is further discussed in Section 4.5.

In conclusion the following aspects for a sound integration of the QPP into the Triple Space System have to be regarded:

- QPP must receive and forward SPARQL queries encapsulated in rd requests
- QPP must only consider 'rd without space' queries.
- Service description of kernels necessary for the QPP’s physical optimization procedures must be stored in a distributed fashion.
- QPP must forward sub-queries and receive results through the distribution manager.

3 State Of The Art

In this chapter we first discuss existing SPARQL query tools based on a small-scale survey on the state of the art. In the next part of the chapter we discuss current state of the art for Distributed RDF Query Engines with decomposition.

3.1 SPARQL Query Tools

SPARQL became a W3C Recommendation [7] in January of 2008, and there are yet relatively few tools that support end users in creating SPARQL queries. There are also few tools that aid in creating and editing queries in other proposed semantic query languages, such as RQL or RDQL [8]. As part of the work in WP3, a small survey was made for the analysis of the existing SPARQL query tools. The results have been used as a ground for decisions in the development of the DSQT.

Most query tools aim to assist the user in developing syntactically correct queries, e.g., by syntax highlighting and auto-completion of SPARQL keywords. There are also some attempts to support the user by providing visual query construction. Visual query tools use graphic notations for the presentation of SPARQL query elements, such as subjects, predicates and objects. They may allow drag-and-drop choosing of these elements from vocabularies. They may use visual notation also for representing other SPARQL features like union graph patterns, result ordering and filtering of results [9]. Potentially these visual tools aid the user by reducing the need to learn the details of the SPARQL query language syntax. They may also help in understanding the schema/ontology of the RDF data that is queried. Some knowledge of SPARQL is usually still needed, since there is a close correspondence between the graphical notations and SPARQL language constructs.

In the following, several SPARQL query tools are briefly described, followed by a table summarizing the main functionalities. Many tools are developed in non-commercial projects, which may be rather small-scale, but there are also commercial products available.

3.1.1 Non-visual Query Tools

TopBraid Composer[1] from TopQuadrant, Inc is a commercial modeling tool for the creation and maintenance of semantic models. It is an editor for RDF(S) and OWL models, as well as a platform for other RDF-based components and services. TopBraid Composer is implemented as an Eclipse[2] plugin and built on top of the Jena framework[3]. TopBraid's SPARQL query editor provides checking of lexical errors as well as keyword highlighting when writing queries. Inferencing can be performed over an OWL DL inference engine Pellet[4], which is included with TopBraid Composer. UPDATE queries are supported as a non-standard extension, based on the Jena ARQ[10] engine. DESCRIBE queries are not supported.

TopBraid is distributed under a proprietary commercial license.

[1] http://www.topquadrant.com/topbraid/composer/
[2] http://www.eclipse.org/
[3] http://jena.sourceforge.net/
[4] http://clarkparsia.com/pellet

Twinkle[5] is a simple user interface that wraps the ARQ SPARQL query engine. The tool intends to be useful both for people wanting to learn the SPARQL query language, as well as those doing Semantic Web development. Twinkle allows querying local files and remote documents as well as saving queries.

Twinkle is distributed under the Gnu Public License, Java source code is available. It is developed by Leigh Dodds.

Morla[6] is a multiplatform editor for RDF documents developed by Andrea Marchesini. Morla supports queries in SPARQL and RDQL languages. Morla's SPARQL Query editor provides syntax highlighting and some auto completion.

Morla is written in C and based on libnxml and librdf libraries. It is a FreeSoftware project released under Gnu public license v2.0.

Sparql Editor[7] from Danny Ayers is a simple HTML form/JavaScript-based tool, which assists in creating syntactically valid SPARQL queries. It allows choosing different SPARQL query constructs from templates and validates the queries.

3.1.2 Visual Query Tools

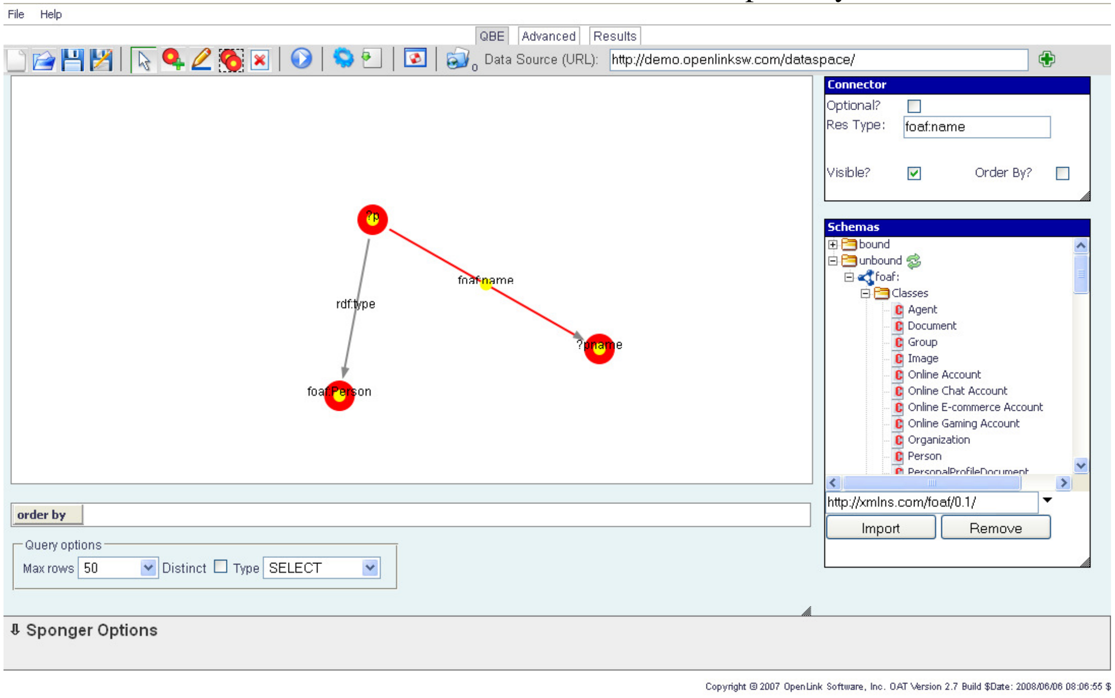
SparqlViz[8] is a plugin for IsaViz[9], which is a visual environment for browsing and authoring RDF models represented as graphs. SparqlViz allows assisted construction of SPARQL queries based on a wizard-like interface, where forms are presented to the user creating a query. According to SparqlViz web site on SourceForge, it is currently being developed by Jethro Borsje and Hanno Embregts for their Bachelor thesis. SparqlViz is written in Java using Jena toolkit.

Nitelight is a graphical tool for SPARQL query construction described in [9] It provides an interactive graphical editing environment that combines ontology navigation capabilities with graphical query visualization techniques. It has been developed as a Java-based prototype, using a combination of Jena and Standard Widget Toolkit (SWT) components. At the time of writing, the software or source code does not seem to be publicly available.

iSPARQL[10] visual query builder is perhaps the most advanced publicly available visual query tool at the moment. A screenshot is iSPARQL is shown in Figure 3.1. It is part of the OpenSource Edition of OpenLink Software's Virtuoso[11]. At core, Virtuoso is a high-performance object-relational SQL database, which supports SPARQL embedded into SQL for querying RDF data stored in Virtuoso's database. The iSPARQL Visual Query Builder supports the user in all SPARQL query result forms (i.e. SELECT, CONSTRUCT, etc.). It also supports the creation of optional graph patterns as well as UNION combinations of graph patterns.

[5] http://www.ldodds.com/projects/twinkle/
[6] http://www.morlardf.net/
[7] http://dannyayers.com/code/sparql-editor
[8] http://sparqlviz.sourceforge.net/
[9] http://www.w3.org/2001/11/IsaViz/
[10] http://oat.openlinksw.com/isparql/
[11] http://virtuoso.openlinksw.com/wiki/main/

iSPARQL has been developed using OpenLink AJAX Toolkit (OAT)[12], which is a JavaScript-based toolkit for browser-independent Rich Internet Application development. OpenLink Virtuoso software is licensed under the GNU General Public License version 2, source code is available from a CVS repository.



Data Source (URL): http://demo.openlinksw.com/dataspace/

Res Type: foaf:name

http://xmlns.com/foaf/0.1/

Max rows 50 Type SELECT

Copyright © 2007 OpenLink Software, Inc. OAT Version 2.7 Build $Date: 2008/06/06 08:06:55 $

Figure 3.1 Query building with iSPARQL query editor

3.1.3 Comparison

Features of the query tools are summarized in the following Table 3.1. These features are partly from requirements identified in section 2.1 and partly from the licensing perspective to see if they allow their inclusion in TripCom project as a 3rd party component. Nitelight is missing from the comparison since the software is not publicly available at the moment. Symbol X means that the feature is implemented by the tool.

	Syntax highlighting	Loading and saving of SPARQL queries	Saving of query results as RDF/XML	Visual query construction	License	Source code available
TopBraid Composer	X	X	X		Proprietary	No
Twinkle		X	X		GNU Public License, v 3	Yes, Java
Morla	X	X			GNU Public License, v 2	Yes, C
Sparql Editor						No
SparqlViz		X	X	X(With forms)	GNU General Public License	Yes, Java
iSPARQL		X		X	GNU Public License, v 2	Yes

Table 3.1 Query tool features

[12] http://oat.openlinksw.com/

3.2 State of the Art for Distributed RDF Query Engines

In the previous Deliverable 3.3 we gave a general overview on engines for SPARQL query processing and on models for the estimation of SPARQL query evaluation costs. As the major contribution of this document is an adaptive concept for distributed SPARQL query processing provided by the QPP, we narrow our focus to methods exclusively tailored for distributed SPARQL querying.

3.2.1 DARQ

Quilitz et al. [8] introduce Distributed ARQ, for short DARQ, a query engine for federated SPARQL queries based on ARQ[9], a SPARQL query engine for single RDF triple stores. DARQ provides transparent access to distributed SPARQL endpoints as if querying a local repository. From an architectural view, DARQ is the mediator component -- similar to the one used in a mediator based information system [12] -- for SPARQL endpoints (see
Figure 3.2). However, different from a MBIS's mediator DARQ does not provide schema integration. To use DARQ the capabilities of each endpoint have to be described by a service description. Such a description comprises the list of RDF property IRIs which are stored at the endpoint, cardinalities for instances, and selectivities. Query processing, i.e. decomposition and distribution of queries, is aligned by these descriptions. Data sources provided by an endpoint as well as a cost estimation based on the statistical information for the endpoints decide the partitioning of a query as well as the assignment of subqueries to endpoints for execution. Service descriptions are only stored locally, i.e. at the machine that runs DARQ.

DARQ focuses query decomposition and distribution exclusively to basic graph patterns and *filtered basic graph patterns* (*FBPG*) [7], i.e. a BGP combined with a FILTER expression. More precisely, DARQ handles graph patterns comprising binary functions of the SPARQL Algebra, i .e. UNION, FILTER, OPTIONAL, GRAPH, by limiting the scope for decomposition and distribution to each FBGP occurring in the graph pattern.

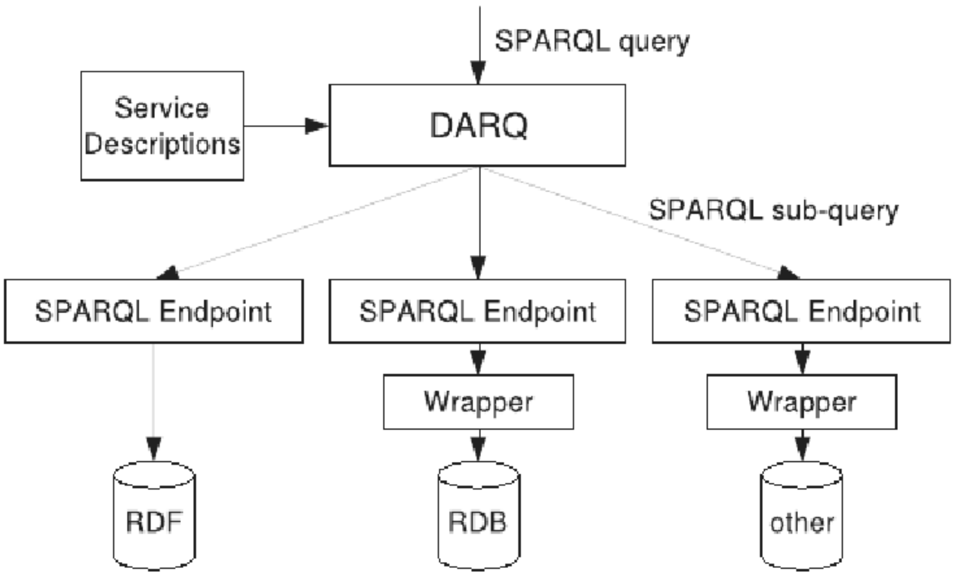


Figure 3.2. DARQ integration architecture

3.2.2 SemWIQ

Similar to DARQ the *Semantic Web Integrator and Query Engine* (*SemWIQ*) [5] works on distributed RDF-repositories with SPARQL support, using endpoint descriptions. Furthermore, it limits decomposition on the occurring BGPs and FBGPs in the graph patterns and stores the service descriptions locally. The latter contain type information provided by OWL ontologies and RDF statistics. This concept is slightly different to DARQ, which states RDF properties, statistics and selectivities for its endpoints. The authors of SemWIQ claim that this variant is, in practice, better maintainable than the one of DARQ, since fewer updates of the descriptions are necessary.

In Figure 3.3, an architectural overview of SemWIQ, explained in detail at section 3 of [5], is given.

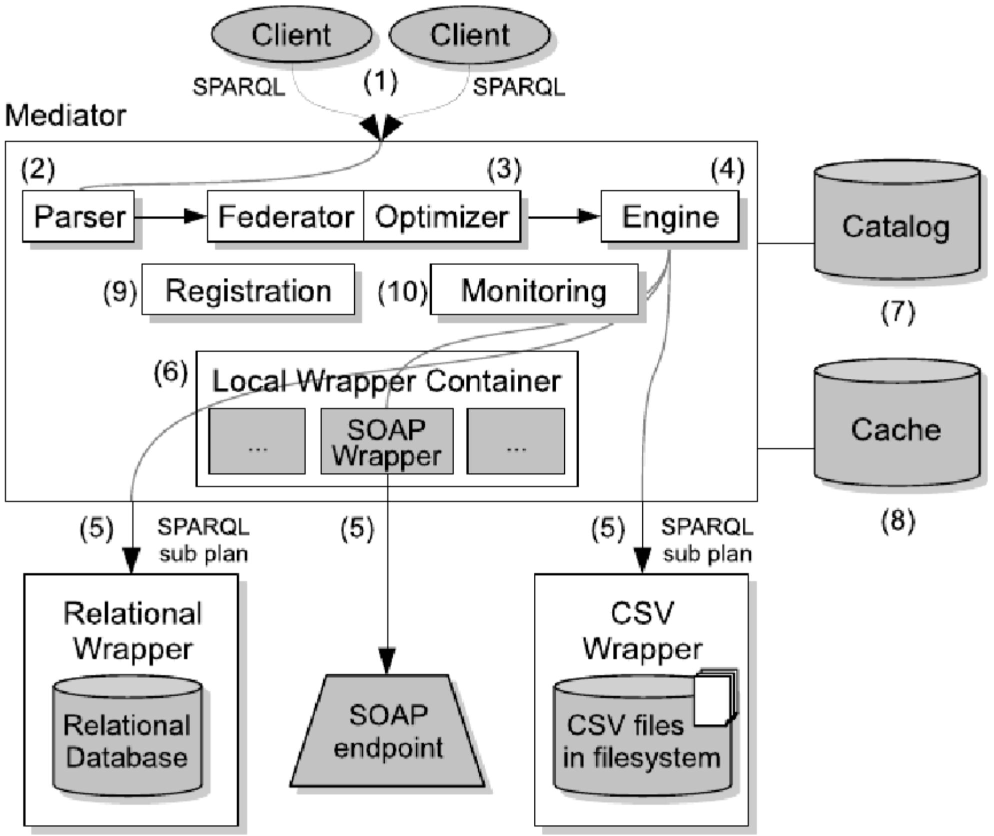


Figure 3.3. SemWIQ integration architecture

3.2.3 FeDeRate

A much simpler approach in comparison to the previous ones is *FeDeRate* [6], a multi database engine, which only enriches local SPARQL processing by the possibility of remote subquery evaluation by named graph patterns.

3.2.4 Comparison

Following the determined requirements on the QPP for interoperability with the Triple Space System (Section 2.2) we determine criteria to evaluate the previously introduced engines regarding their compliance with and utilization of the Triple Space System's premises. A summary of the results of our evaluation including a preview of the QPP's properties is given in Table 3.2.

As mentioned earlier the Triple Space System's architecture not only offers the opportunity to execute queries in a distributed fashion, but also offers starting points for significant physical optimization. Regarding the latter RDF statistics for each space can be provided, and hence based on this data execution costs can be estimated. In turn, the actuation of an optimized query execution plan can be based on these costs (for more details, see Section 4). Consequently, we highly consider, if an engine spends attention to execution costs when sending sub-queries to distinct kernels for evaluation (Table 3.2). Furthermore, we have neither a global ontology nor ontology integration mechanism existing or planned in the Triple Space System. Therefore we verify that the cost evaluation and sub-query distribution is solely based on RDF-Statistics. Apart from that we check if the engine supports a distributed storage of service descriptions, such as RDF statistics, evidently mandatory for the distributed architecture of the Triple Space System.

query engine	decomposition and distribution based on				service description storage
	based on evaluation costs	based on RDF statistics	based on selectivity	based on types	
DARQ	yes (very simple approach)	yes	yes	no	local
SemWIQ	yes (very simple approach)	yes	no	yes	local
FeDeRate	no	no	no	no	local
QPP	yes	yes	no	no	distributed

Table 3.2: Comparison of distributed SPARQL query engines

Our analysis showed that neither of the engines is fully capable of coping with the special opportunities and requirements of the Triple Space System. Especially none of them is prepared to deal with distributed service descriptions. Furthermore, there are no optimization techniques in place that capitalize on the very low recall rate for query evaluation mandatory for the Triple Space System. Despite these shortcomings, DARQ could at least provide some incipient inspirations for our research work on the QPP. Specifically, we took the following notions into account:

- DARQ provides a transparent query decomposition and distribution mechanism.

That is, remotely executed subqueries are dispatched as ordinary queries to the remote SPARQL endpoints; these endpoints are not aware that they are executing a part of bigger query.

- DARQ's service descriptions rely on statistical and lexicographical characteristics of the RDF data stored at a local database but not on ontologies about that data.

- DARQ supports a cost-based query execution. The used cost estimation provides a very rough approximation of the actual execution time based on the size of data to ship to and from remote endpoints during query execution.

However, DARQ's ideas could only provide a very loose basis for a few detached parts of the QPP concept since, likewise, DARQ suffers from the same limitations mentioned precedingly. The severity of these insufficiencies will become more evident during the course of the upcoming sections. Most significantly, we will learn that the QPP provides unprecedented features as time-preserving optimization for receiving a small subset of the perfect recall result for a query and an efficient way for retrieval of distributed metadata.

4 Query Preprocessor Concept

A core idea of RDF and the Semantic web is to link different sources of machine-processable information in a way that allows automatic combination of different data. However, in the current prototype of the TripCom Kernel, a SPARQL query, transmitted as template of a 'rd without space' call, yields a result only if an according set of triples is located within a single space. More in line with the inherently distributed nature of the Triple Space system, the QPP improves upon this situation by splitting SPARQL queries into subqueries and querying several kernels at once in order to find answers that can be assembled from triples located in several spaces. Thereby all actions are executed with respect to logical properties of the query, physical characteristics of the distributed environment and database statistics. Apart from that, subqueries are transparently solved, since they are dispatched to spaces as query templates of 'rd with space' calls.

Recall [13] is our metric to evaluate the QPP's information retrieval efficiency. Here, it measures, in percent, the amount of solution mappings the QPP returns in comparison to all possible solution related to a query. For the latter the query is matched against the integrated data set of all data source spaces the DHT provides for triple patterns in the query. A recall of 100% means that all possible solutions are returned and is called *perfect*. To prevent ambiguousness of the searched data base we presume that no changes of data sets take place in any space of the Triple Space System during the time interval a query is processed.

In TripCom the recall for a query over multiple kernels is allowed to be non-perfect. This circumstance grants us to pick just a subset of all available data sources for a sub-query evaluation. Therefore, we are free to decide which optimization goal, low costs or high recall, we prioritize while choosing the query execution plan. In this context, we have to consider, that the Triple Space System conceptual design is fundamentally inspired by Linda tuple spaces, for which a single element from all possible solutions is always sufficient. Considering these aspects, we will strongly align our optimization measures in the QPP to return only a few of all solution bindings for a SPARQL query to be solved. As we shift the focus of our optimization away from high recall we aim for a major improvement in the execution time of queries.

Another significant characteristic is the distributed metadata storage and retrieval. In contrast to DARQ, metadata have not necessarily to be stored locally at the kernel, where the query was initiated. Each kernel creates and stores its own metadata by the TS Adapter, and, in addition, can retrieve metadata from remote kernels. This difference gives the QPP a formidable advantage for the highly distributed, large-scale characteristics of the Triple Space system.

As mentioned earlier, the TripCom kernel encapsulates each SPARQL query as in a rd operation as 'query template'. Consequently, this convention must be considered by the QPP for each SPARQL query received or forwarded. Moreover, since the QPP, by concept, does not target a concrete kernel as data source, only 'rd without space' queries are handled by the QPP.

Before we explain the partial concepts of the QPP in-depth, we present an overview of the architectural workflow. The QPP processes a query q in four stages:

1. **Parsing:** Initially, query q, given as the SPARQL query string, is parsed by the query engine *ARQ* [10] into an operator tree, i.e. a logical, intermediate tree representation of SPARQL queries corresponding to the parenthesis structure of the abstract SPARQL query term reassembling q. As stated in Section 3.2.1, ARQ is a SPARQL query engine for operating upon a single RDF triple store.

2. **Logical Optimization:** Second, query q is logically optimized by transforming it to an equivalent but more cost efficient form per the application of rewriting rules (see Section 2.2).

3. **Query Decomposition:** As third step query q is decomposed into subqueries with respect to the metadata received from the Distribution manager on kernels with applicable data sources (see Section 4.1.3 and 4.4).

4. **Physical Optimization and Query Execution:** Fourth, the most auspicious query plan for q --- chosen by estimating the physical costs based on RDF statistics and looking up appropriate spaces as data sources on the Distribution Managers DHT ---- is executed. More precisely, actions on physical optimization and query execution are interwoven for better adaptiveness: A partial query plan, i.e. a set of subquery, is sent to the Distribution Manager which forwards them to the targeted kernels. When the answers arrive the Distribution Manager sends them to the QPP which constructs the partial answer for the overall query q and the partial query plan to be processed next. Eventually, after a finite number of repetitions, a set of solution mappings for q is achieved (see Section 4.5).

4.1 Metadata for Kernels

In this section we describe the metadata about a kernel which is relevant for query planning, i.e., metadata necessary for the QPP's third and fourth stage (see introduction of Section 4). This information comprises physical characteristics in the form of database statistics. It is maintained and stored for each kernel by its TS Adapter except for the triple pattern index managed by the DM.

4.1.1 Triple Pattern Indices

The Distribution Manager maintains an index, in form of a distributed hash table (DHT), with triple patterns as keys, each mapping to a list of Space URLs, that provide binding for the key triple pattern. The QPP will use this index when assigning subqueries to kernels for dispatch. For a triple pattern t we will denote its value in the Distribution Manager's index as $sources(t)$. Furthermore, for a set of triple patterns T, we refer to the set

$$sources(T) := \bigcup_{t \in T} sources(t) \tag{1}$$

as *data sources* for T.

4.1.2 Statistical Database Information

The following statistical values about the structural state of a space D embodied in the kernels TS Adapter have to be provided (let $?s, ?p, ?o \in RDF - V$):

1. The total number of triples, referred to as n_D.
2. For each bound predicate p the number of triples for a triple pattern $(s?, p, o?)$, referred to as $n_D(p)$.

For a local space A one can find the statistical metadata in the space with URI '<URI of A>/metadata'. For a remote space one can find the cached statistical data, if available, in the local space with URI '/remote-metadata/<name of remote space>'.

4.1.3 Retrieval of Metadata from Remote Kernels

A significant problem in the distributed environment of the Triple Space System is the exploration of an efficient method to acquire the metadata of the remote kernels, that are potential data sources for the distributed query execution. These are kernels, that can be found in the DM's index for at least one triple pattern of the BGP to compose. In general, several paradigms can be used for this purpose. The following approaches were analyzed:

- a central database for metadata
- a distributed database or a DHT for metadata
- flooding the kernels without metadata retrieval
- asking the kernels for the metadata

A central database, distributed database or DHT would mean, that every kernel had to send periodic updates to the metadata database, which would cause a lot of net load. In addition to that, a central database would break the distributed system paradigm of the Triple Space System, while DHT updates yield logarithmic worst-case costs for lookups and updates with respect to the kernel number. Just flooding all data source kernels without the retrieval and consideration of their metadata would cause a lot of net load and kernel load in the Triple Space System every time a query is triggered. Hence, often we only want to ask a very small subset of all data source kernels for the solution of a query, as we will see in Section 4.5. As result of this consideration we favor a compromise, where essentially the kernel initiating the query asks a selection of remote data source kernels for their spaces' RDF statistics directly, executed at step 2(b)v of the algorithm in Section 4.5.

4.2 Constraints on SPARQL queries and Kernel Configurations

For the utilization of the QPP benefits, we have to accept some mandatory constraints, though:

- Constraints on SPARQL queries
    - o Since the QPP uses predicates to decide where to send triple patterns, only triple patterns with constant predicates may occur in SPARQL queries.

  - Joins using blank nodes are not supported. If a join operation finds a blank node an exception will be thrown.
  - The operators DESCRIBE and GRAPH are not supported

- Constraints on the Triple Space System configuration
  - All security facilities are deactivated. Consequently, we assume that there are no malicious actors (kernels, clients).
  - All self-organization facilities are disabled.

4.3 Logical Optimization

Generally speaking, logical query optimization performs rewriting of queries while preserving their semantics to achieve less costs for execution. Mostly the structure of a query is simplified and reduced in length, thus less operation have to be processed.

In our case, before we split each BGP embodied in a query q into subqueries, we apply on q best practice rewriting rules to cluster triples and filter operators to BGPs, which will improve the result of the upcoming decomposition process. For this we will use the same rules as DARQ applies for logical optimization. A typical example for such a rule is presented in Section 3.3 of [8].

4.4 Query Decomposition

After a SPARQL query q is translated by the ARQ parser into its corresponding operator tree, we detect for each BGP b in q the data sources for its triple patterns and, subsequently, partition b into subgraphs, which then can be dispatched as sub-queries to spaces later on (see Section 4.5).

First of all, we define for a directed weakly connected planar graph $G = (V, E)$ the term *subgraph cover* as a set C of weakly connected, edge-disjunctive subgraphs of G, such that each edge in E is also an edge of one and only one subgraph in C. Furthermore, each subgraph possesses a data source space (determinable by the DM's DHT).

Let $size$ denote the referential edge-number for subgraphs in a subgraph cover. In our evaluation, we chose $size = 3$ as default. On each BPG b in q we concurrently run the following steps to determine a subgraph cover C:

1. Determine the weakly connected components of b.

2. For each weakly connected component c concurrently carry out the following actions:
   (a) Determine via the DM's DHT the data source spaces for each triple pattern in c
   (b) If a data source space exists for all triple patterns in c, then $\mathsf{C} = \{c\}$ and terminate. Else determine a subgraph cover C, such that each subgraph has a number of edges e with $e \leq size$. This is determined via a greedy BFS-/DFS-traversion of c and data source intersections of the triple

patterns in c.

Later on, the algorithm presented in Section 4.5.1 will send each BGP b in C, embedded in the SPARQL query CONSTRUCT {b} WHERE {b}, to a kernel with matching bindings. Furthermore, for each BGP b, encompassed by a filter constraint, we consider to add filter expressions to the subqueries of b: If all variables in b's constraint are also present in subquery s, we add this filter to s. If the constraint of b can be split, i.e. if the constraint is a disjunction, we add the partial constraint holding the same variables as s. Filters which concern variables for more than one subquery and cannot be divided and, therefore, have to be applied locally by the QPP.

As a reminder, we want to underline at this time that each subquery shipped to a space, is sent as a template in a 'rd with space' request. Consequently, the QPP of the kernel that hosts the target space is not at all concerned by this request. This is due to the fact that the QPP, if activated, exclusively processes 'rd without space' invocations.

4.5 Physical Optimization and Query Execution

Until this point we received a subgraph cover for each BGP in query q. Now we describe the technique to efficiently evaluate the subqueries for each BGP and, eventually, the overall query q. Thereby, the query execution and the physical optimization of the query plan are tightly interwoven with each other. Their interplay is concisely described in the following algorithm. As preliminary to this algorithm, we have to expose the join implementation we use for combining the results for sub-queries of a BGP. For this conjunction we use *indexed nested-loop joins* [2] and, in particular, *bind joins* [3]. Basically, a bind join is a nested-loop join where intermediate results from the outer relation are passed to the inner to be used as a filter. This approach has two advantages in comparison to techniques primarily suited for joins on a single machine, like (indexed) nested-loop join, hash join or merge-sort join [2]. On one hand, the amount of data shipped across the network is extremely reduced, if the unbound subquery would return a large result. On the other hand, the computation of the join is more decentralized which means that the load of recombining the intermediate results is distributed over several machines instead of solely be dispatched by the machine which initiated the query.

As mentioned earlier, the optimization of the QPP is tailored towards a time-efficient retrieval of a small subset of the overall set of solution bindings for q with respect to the integrated global database. Regarding these premises, the evaluation of a BGP b in q is organized into *bind-join clusters*, in which subgraphs of b are joined by bind-joins with the aid of the cost estimation and the DM's DHT. This join algorithm reduces the risk of selecting data sources as join partners that offer no matching for subgraphs or selecting much more results than necessary to find at least one binding for q. After each bind-join-cluster of b is evaluated the clusters can be joined itself and, thus, will yield a solution for b.

Subsequently, we will explain in detail how a query is evaluated in the context of bind-join clusters. Subsequently, we define the cost function used to choose data sources for subgraphs within bind-join clusters.

4.5.1 Query Evaluation and Bind-Joins Clusters

After we discovered a subgraph cover in Section 4.4 for each weakly connected component of each BGP in query q, we now proceed to the evaluation of q. First, we describe the steps that are necessary to receive the solutions for the BGP in q. Subsequently we describe the constructions of the overall answer for q based on the results for the BGPs.

On each BPG b in q we run the following steps to determine its solution:

1. Define *size* of type integer and set $size := k$ with $k > 0$. In our evaluation we chose $size = 3$ as default.

2. For each weakly connected component c with subgraph cover C concurrently carry out the following actions:
   (a) Group the subgraphs in C to disjunctive sets, denoted as *bind-join clusters*. Each bind-join-cluster must embody $\leq size$ elements. These clusters are determined via a greedy BFS-/DFS-traversion of c', which is the graph derived from c by substituting each subgraph in C with an individual node.
   (b) For each bind-join cluster *bjc* concurrently carry out the following steps:
      i. Define the variables *visited-subgraphs* of type set of subgraphs, *active-subgraphs* of type subgraph and *bjc-solutions* of type solution mappings. Initiate $\text{visited} - \text{subgraphs} := \varnothing$.
      ii. Initiate *active-subgraph* with a subgraph in *bjc*, which has a minimum number of variables. If more than one such a subgraph exists, randomly choose among these a subgraph with a minimum number of spaces.
      iii. Set $\text{visited} - \text{subgraphs} := \text{visited} - \text{subgraphs} \cup \{\text{active} - \text{subgraph}\}$.
      iv. Select a data source space of *active-subgraph* for evaluation: If up-to-date RDF statistics (see Section 1.3) are locally available for at least half of the data source spaces for *active-subgraph*, choose a space D, which has median estimated evaluation costs, i.e. median $costs_D(\text{active} - \text{subgraph})$. Else, choose a random data source space.
      v. Concurrently query the RDF statistics of l randomly chosen data spaces of *active-subgraph* without up-to-date locally store metadata. In our evaluation we used $l = 4$.
      vi. Send *active-subgraph* as subquery (see Section 4) via the DM to the selected space D for evaluation.
      vii. For each received binding bd from D concurrently do the following steps:
         A. If $visited - subgraphs = bjc$, set

$\text{bjc}-\text{solutions} := \text{bjc}-\text{solutions} \cup \{bd\}$ and cancel the following steps of this thread.

B. Determine in bjc a subgraph *sg*, that is
   - adjacent to *active-subgraph*,
   - is not an element of *visited-subgraphs*,
   - and its instantiation sg' caused by bd has at least one data source space.

C. If $sg \neq null$, create a new bind-join cluster bjc', which differs from bjc only by the fact that *active-graph* is substituted by $\text{active}-\text{graph}'$, which is the instantiation of *active-graph* caused by bd. Set $bjc := bjc'$, $\text{active}-\text{subgraph} := sg'$ and go back to step 2(b)iii.

D. If $sg = null$, cancel the algorithm globally. Return the empty set as result value to the algorithm caller.

(c) Join the solutions for the bind-join clusters of c. These joins are done via indexed nested-loop joins. Each bind-join cluster bjc has its solution stored accordingly at *bjc-solutions*.

3. Union the solutions for each weakly connected component of b.

Until now, we have just determined the solutions for each BGP in q. In the following, we will construct the answer for q. We define a unique named graph $g_i = (iri_i, triples_i)$ for each BGP b_i of q where $triples_i$ is the union set of triples retrieved as solutions for b in the previous paragraph. Now we can rewrite q in the way that each BGP b_i is replaced by 'GRAPH g_i $\{b_i\}$'. Thus q can now be inserted into ARQ while we choose $(\emptyset, g_1, \ldots, g_n)$ as RDF dataset.

4.5.2 Evaluation Costs and Selectivity

As we have seen, evaluation costs are necessary to make cost efficient data source choices in step 2(b)iv of the algorithm in Section 4.5.1. In general, the term *evaluation costs* is ambiguously defined as it can stand for a variety of different physical metrics [4], most commonly, query execution time or query execution steps. We define it as a relative execution time value based on the result cardinalities for subqueries. This value is motivated by the fact that in distributed query engines data transfer between nodes (machines) are the major factor on the query execution time. In our case, the costs express the relative time for the shipping subqueries and subquery results between spaces on the same kernel or, more importantly, between two kernels. Additionally, also the time for joining subquery results are taken into account. Accordingly, we define the costs function for a subgraph g of a BGP b with data source space D as follows

$$costs_D : TP^2 \rightarrow \mathbb{R} \tag{2}$$

$$costs_D(g) = \prod_{t \in g} card(t) \cdot sel \tag{3}$$

where TP denotes the set of all valid triple patterns, sel the generic estimate of the join selectivity (0.5 as default in our evaluation) and $card : TP \rightarrow \mathbb{R}$ the function, which assigns a triple pattern $(s, p, o) \in TP$ the cardinality according to the statistical data, that is provided for p regarding D.

5 IMPLEMENTATION

In this chapter we give an overview of the implementation of the Distributed Semantic Query Tool and Query Preprocessor.

5.1 SPARQL Querying

A semantic query tool for distributed queries in the triple space has been developed as a software deliverable of the work package 3. The tool supports developers by showing metadata about spaces that reside on Tripcom kernels, and by showing information about RDF vocabularies that are used in those spaces. It also aids developers in constructing syntactically valid SPARQL queries. Here we discuss the key points of the implementation.

5.1.1 Architectural Overview

Distributed Semantic Query Tool has been implemented as components which are external to the Triple Space kernel. The user interface has been implemented with Google Web Toolkit (GWT), which is an open source framework for developing Ajax style applications with a subset of Java programming language[13]. Applications developed with GWT work in all major browsers (IE, Firefox, Mozilla, Safari, and Opera), which was a major reason for choosing it as a development platform. In the following, GWT-based user interface component is referred to as client side component.

Client side component interacts with a server side component. The server side component consists of Java servlets, which are needed mainly for communication with Triple Space. The server side component interacts with the kernel with messages generated by the TSClient component. Communication between client and server-side subcomponents has largely been implemented by using GWT Remote Procedure Call (RPC) mechanism.

5.1.2 Communication Between Components

An overall view of the communication between different subcomponents is depicted on Figure 5.1. When developer begins querying Triple Space, client side subcomponent requests a list of subspaces and related spaces of a certain user specified space from server side subcomponent, which uses TSClient to contact the Triple Space. Returned list is shown to the user as a tree of spaces. The developer selects the spaces which (s)he wants to query. SPARQL query is then formed requesting RDF vocabularies that are used in the spaces that the user wants to query. Information about predicates; their ranges, as well as their labels and comments, if available, is returned to the client side subcomponent. For this to work, the vocabularies must be stored in the same space or in some other dedicated space.

SPARQL query created by the developer can be sent to the server side component to get information about query cost, which may help in reformulating the query. Finally the query is sent to the Triple Space for evaluation and result is returned to the user via server-side component.

[13] http://code.google.com/webtoolkit/

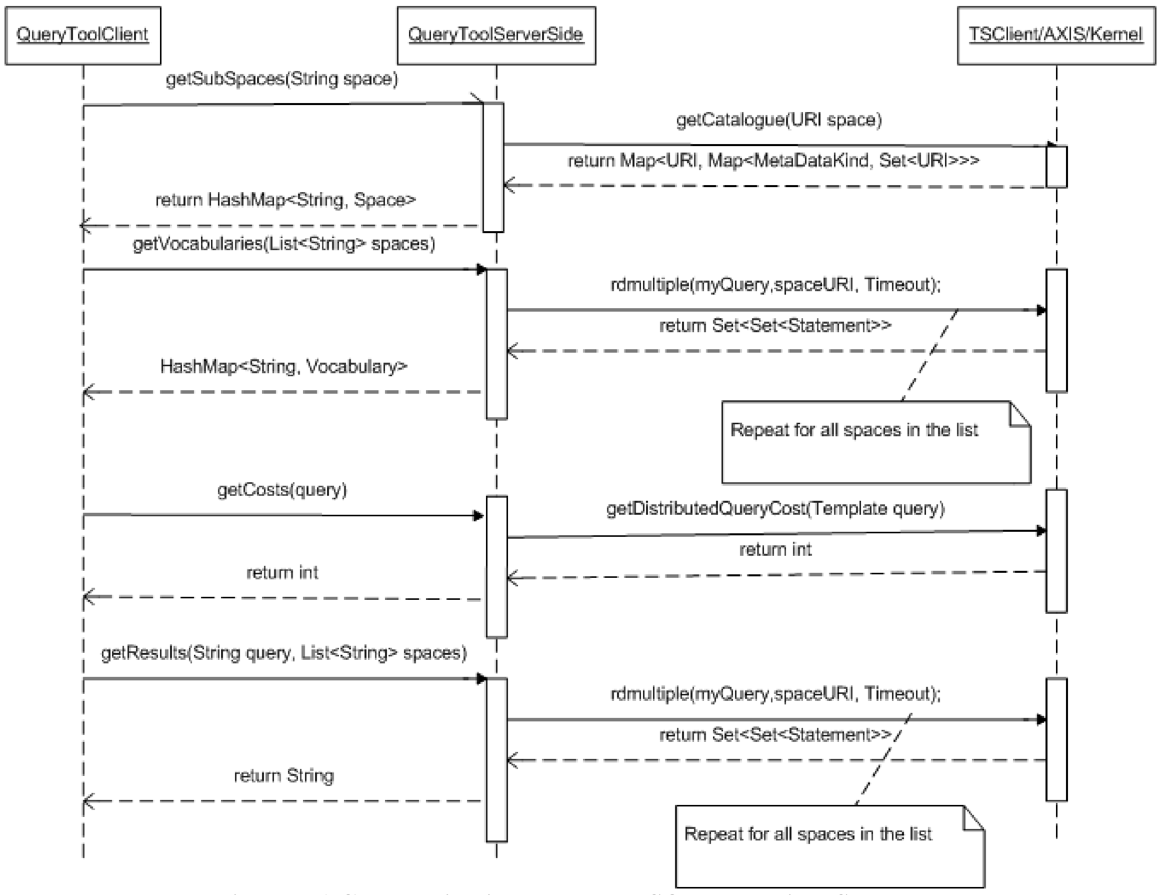

QueryToolClient
QueryToolServerSide
TSClient/AXIS/Kernel
getSubSpaces(String space)
getCatalogue(URI space)
return Map<URI, Map<MetaDataKind, Set<URI>>>
return HashMap<String, Space>
getVocabularies(List<String> spaces)
rdmultiple(myQuery,spaceURI, Timeout);
return Set<Set<Statement>>
HashMap<String, Vocabulary>
Repeat for all spaces in the list
getCosts(query)
getDistributedQueryCost(Template query)
return int
return int
getResults(String query, List<String> spaces)
rdmultiple(myQuery,spaceURI, Timeout);
return Set<Set<Statement>>
return String
Repeat for all spaces in the list

Figure 5.1 Communication between DSQT and Triple Space

5.1.3 Overview of the User Interface

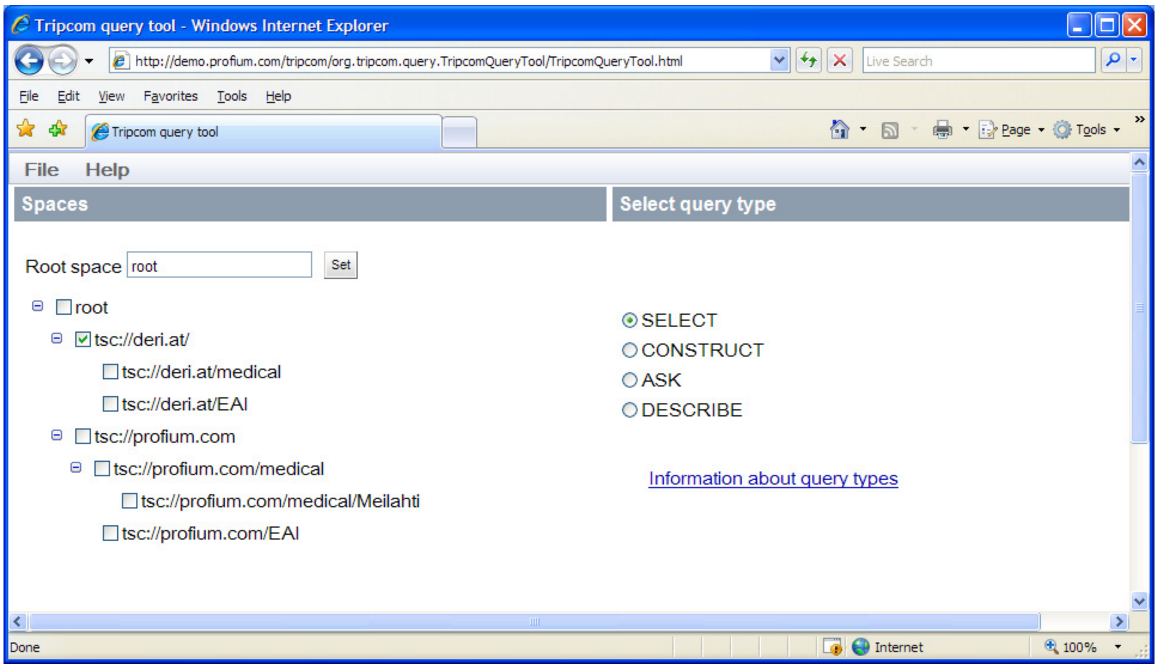
In this subsection, a brief overview of the user interface is given. More complete description of the functionality can be found from the user guide of the query tool.

The user interface is based on form-based templates, which aid the developer in creating syntactically correct SPARQL queries. Slightly similar approach has been used by SPARQLViz[14] described in section 3.1. Approach was chosen, since form-based templates were considered easier to use by those who have limited SPARQL experience, compared to the completely graphical construction of queries. On the other hand, form-based query interface was easier to implement with the technology and resources available for the task.

A developer starts the query construction by selecting the spaces to query. The tree on the left side of the screen in Figure 5.2 shows the subspaces of the space that the user has specified as a start point of the query. The developer will then see the related spaces by moving the mouse over the address of a space at the tree. Spaces may have related, similar or see also relations.

It must be noted that only CONSTRUCT queries are supported by Triple Space, although the tool supports creating SELECT, ASK, and DESCRIBE queries.

[14] http://sparqlviz.sourceforge.net/


Tripcom query tool - Windows Internet Explorer
http://demo.profium.com/tripcom/org.tripcom.query.TripcomQueryTool/TripcomQueryTool.html
File Edit View Favorites Tools Help
Tripcom query tool
File Help
Spaces
Select query type
Set
Information about query types
Done
Internet
100%

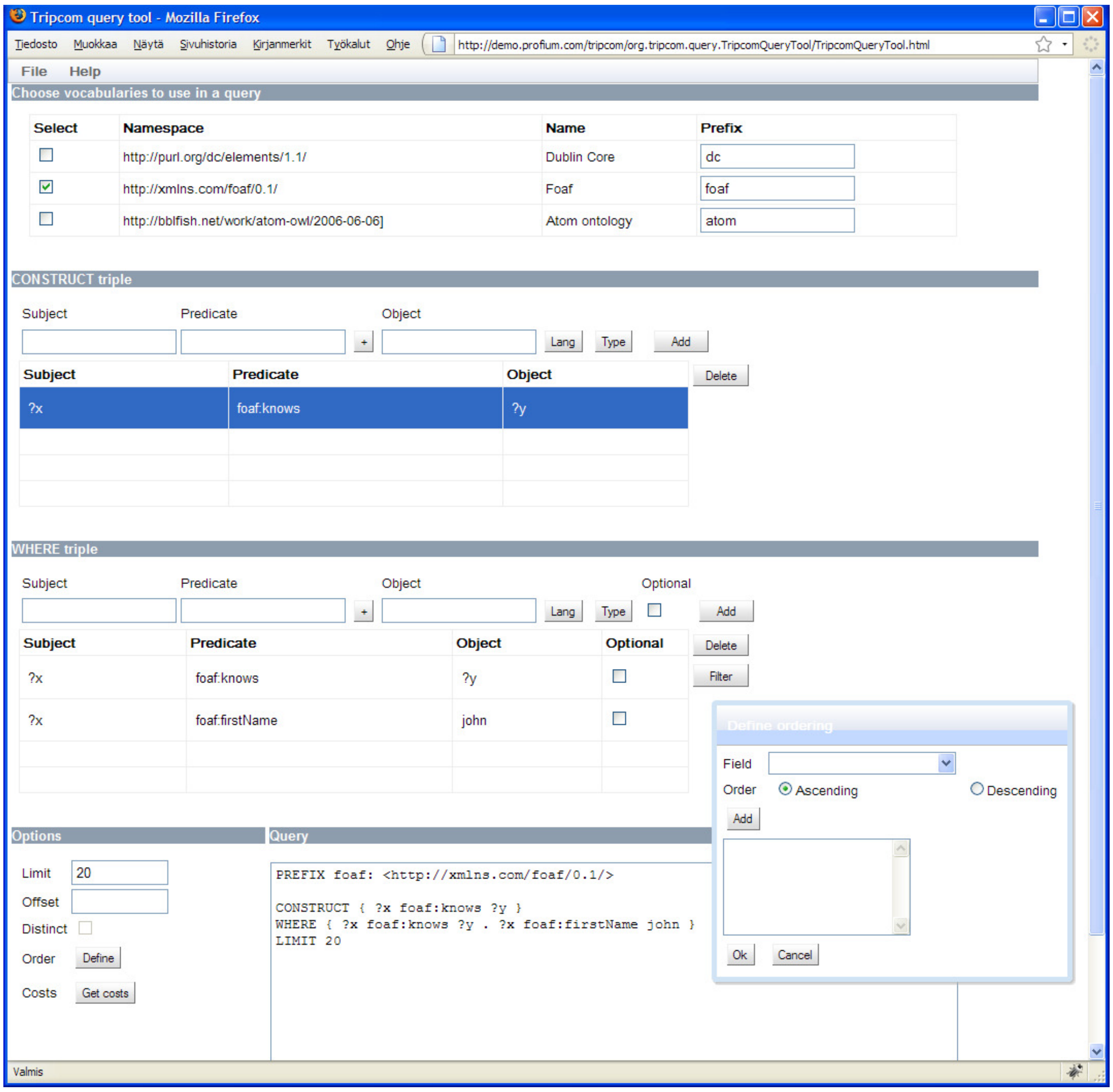
Root space root

- [ ] root
  - [✓] tsc://deri.at/
    - [ ] tsc://deri.at/medical
    - [ ] tsc://deri.at/EAI
  - [ ] tsc://profium.com
    - [ ] tsc://profium.com/medical
      - [ ] tsc://profium.com/medical/Meilahti
    - [ ] tsc://profium.com/EAI

[✓ SELECT] CONSTRUCT ASK DESCRIBE

Figure 5.2. Selecting spaces

The developer can construct a query by using a form depicted on Figure 5.3. The tool shows a list of vocabularies that are used in the space(s) that were selected by the developer on previous screen. The query is constructed by adding triple patterns using the form and by defining possible modifiers and filters. The tool assists the developer in creating semantically meaningful queries by showing properties and their ranges from the vocabularies used in the space(s) to be queried.


Tripcom query tool - Mozilla Firefox
Tiedosto Muokkaa Näytä Sivuhistoria Kirjanmerkit Työkalut Ohje
File Help
Valmis

http://demo.profium.com/tripcom/org.tripcom.query.TripcomQueryTool/TripcomQueryTool.html

Choose vocabularies to use in a query

Select	Namespace	Name	Prefix
☐	http://purl.org/dc/elements/1.1/	Dublin Core	dc
✓	http://xmlns.com/foaf/0.1/	Foaf	foaf
☐	http://bblfish.net/work/atom-owl/2006-06-06]	Atom ontology	atom

CONSTRUCT triple

Subject Predicate + Object Lang Type Add

Subject	Predicate	Object
?x	foaf:knows	?y

Delete

WHERE triple

Subject Predicate + Object Lang Type Optional ☐ Add

Subject	Predicate	Object	Optional
?x	foaf:knows	?y	☐
?x	foaf:firstName	john	☐

Delete Filter

Define ordering

Field
Order [✓ Ascending] Descending
Add
Ok Cancel

Options

Limit 20
Offset
Distinct ☐
Order Define
Costs Get costs

Query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?x foaf:knows ?y }
WHERE { ?x foaf:knows ?y . ?x foaf:firstName john }
LIMIT 20
```

Figure 5.3. Query construction

When the query is sent to the Triple Space, the results are shown on the next screen. The results and the SPARQL query can be saved locally.

5.2 Query Preprocessor

In the following, we enumerate the criteria most crucial for technical implementation of the QPP.

5.2.1 Integration of the QPP in the Kernel Architecture

The QPP must be implemented as a subcomponent of the Distribution Manager. It can be activated via the Distribution Manager's system properties file.

5.2.2 Communication with the Semantic Query Tool

A new TS API operation 'getDistributedQueryingCosts' has been introduced. This operation takes a SPARQL query string and returns the estimated abstract execution costs for query processing, a positive real (Figure 5.1).

Remark: The cost estimation for distinct parts of an inputted query is not possible, since until a query arrives at the cost estimation stage in the QPP it is transformed by rewriting rules. Therefore a correspondence between parts of the original query structure and their evaluation costs is not generated at any time. Additionally, the initial query structure might yield much more costs than the optimized rewritten one.

5.2.3 Metadata Format, Storage and Retrieval

The RDF Statistics for a Space D in a kernel's TS Adapter, explained in Section 4.1.2, should be stored in D's local subspace '<URI-of-D>/metadata'. For a remote space one can find the cached statistical data, if available, in the local space with URI '/remote-metadata/<name of remote space>'. Consequentially, metadata from local or remote spaces can be retrieved by using the standard TS API rd-operations. The formatting of these data is specified by the OWL-Lite ontology given in Listing 5.1.

```
<?xml version="1.0"?>


<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY owl2xml "http://www.w3.org/2006/12/owl2-xml#" >
    <!ENTITY metadata "http://www.tripcom.org/metadata.owl#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>

<rdf:RDF xmlns="http://www.tripcom.org/metadata.owl#"
     xml:base="http://www.tripcom.org/metadata.owl"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl2xml="http://www.w3.org/2006/12/owl2-xml#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:metadata="http://www.tripcom.org/metadata.owl#">
    <owl:Ontology rdf:about=""/>
    <!-- *** Data properties *** -->

    <!-- http://www.tripcom.org/metadata.owl#hasCardinality -->

    <owl:DatatypeProperty rdf:about="#hasCardinality">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="#ConstantPredicate"/>
        <rdfs:range rdf:resource="&xsd;int"/>
    </owl:DatatypeProperty>


    <!-- *** Classes *** -->

    <!-- http://www.tripcom.org/metadata.owl#ConstantPredicate -->

    <owl:Class rdf:about="#ConstantPredicate">
        <rdfs:subClassOf rdf:resource="&owl;Thing"/>
    </owl:Class>

    <!-- http://www.w3.org/2002/07/owl#Thing -->

    <owl:Class rdf:about="&owl;Thing"/>
</rdf:RDF>
```

Listing 5.1: Ontology for RDF statistics

5.2.4 3rd party libraries

For implementation we primarily used Jena[11] and ARQ[10] as 3rd party libraries. The current stable version of Jena and ARQ is 2.5.6 and 2.4, respectively. ARQ is implemented on top of the *Jena framework* (http://jena.sourceforge.net/), a Java framework for building Semantic Web applications. Jena provides the means for handling RDF, RDFS and OWL data. One can find ARQ's source code API description at http://jena.sourceforge.net/ARQ.

6 EVALUATION OF THE QUERY PREPROCESSOR

This chapter describes results of the evaluation of the Query Preprocessor.

6.1 Global Evaluation Schema

In this section we define the universal conditions on which all specific evaluation scenarios are based on. Therefore, we specify the Triple Space environment configuration as well as mandatory constraints on distributed data population, query evaluation and query patterns.

First, we want to mention, that at the time we executed the evaluation tests some kernel functionalities that are crucial for the QPP's operations (see Section 4), were not fully implemented. Thus, for evaluation we used a light-weight version of the TripCom kernel, providing all necessary features the QPP depends on, instead of the baseline TripCom kernel.

6.1.1 Triple Space System Configuration

Hardware Setup
An Ethernet LAN (1 Gbit/s) consisting of 4 physical host machines (CPU: Pentium D, RAM: 2 GB, OS: Debian Linux Lenny) was deployed, each running a single TripCom kernel.

Kernel Configuration
We presume that the Security Manager is disabled. In addition, all self-organization facilities have to be deactivated. This means, in particular, that no spaces are distributed across several kernels, and no subspace is located on a different kernel than its superspace.

Light-weight Kernel
The light-weight TripCom kernel comprises the components crucial for the QPP, i.e. TS API, DM and TS Adapter. The TS API is accessible via the TS Client. The TS API and the DM in conduction provide the same functionality for rd, out, in (with or with out a given Space) as their original counterparts. The DM embodies the QPP as subcomponent. Consequently, the QPP can forward and receive 'rd without Space' operations directly through the DM. The TS Adapter extends the original TS Adapter by offering RDF statistics for each space hold by the kernel (see Section 1.3). Furthermore, it uses Openlink Virtuoso (with allowed memory usage 2GB) as integrated RDF triple storage and local SPARQL endpoint.

6.1.2 Data Population and Query Evaluation

Distributed Data Population
Data distribution across kernels is aligned with the structure of the test query and vice versa, in the sense, that (almost) all kernels can participate in the distributed evaluation of the query. We will demonstrate this issue more precisely by giving examples in Section 6.2.

Query Evaluation
It has to be clarified, that the query evaluation in the QPP takes place exclusively on the set-theoretic level of RDF data, i.e. RDF triples, equivalent to the proceeding of a pure SPARQL query engine. That means that no reasoning based on ontologies is done by the QPP. Evidently, no local ontologies as well as no ontologies from remote kernels have to be retrieved, integrated or considered in any way by the QPP.

6.1.3 Common Query Patterns

As mentioned earlier, the QPP's query evaluation process limits the scope of query decomposition and distributed evaluation to each of the BGPs and FPBGs occurring in the graph pattern of a query. Thus, for evaluation it is sufficient to choose queries whose WHERE-clause holds just a single BGP or FBPG, in which, as stated before, each triple pattern must have a constant predicate. Thus, this graph pattern, in the following referred to as *gpattern*, consists of n RDF triples $t_i = (s_i, p_i, o_i) \in (V \cup I \cup L \cup B) \times IRI \times (V \cup I \cup L \cup B)$ and a filter expression *expr*, which is empty, if *gpattern* is a BGP (here V denotes the set of RDF node variables, I the set of IRIs, L the set of RDF literals and B the set of blank nodes; for more information see [7]). At the last section we already pointed out, that the choice of the triples in *gpattern* and the distribution of data across kernels have to be adjusted, such that virtually every kernel can participate in the query execution. Though, we don't restrict the graph type of *gpattern*, i.e. we don't care, if *gpattern* resembles a tree --- for instance, a left-deep tree or bushy tree -- or any other kind of planar graph.

6.2 A Health Case Scenario

Following the approach of use case 4 in Deliverable 8B.1, we specify here a health case scenario for evaluation.

6.2.1 Storyboard and Queries

A patient wants to find a doctor that fulfills his requirements, i.e. the physician must provide the treatments the patient requires, should be located near to the patients place of residents and should accept the patients insurances. Furthermore, the patient wants to be sure that the treatments provided are covered by his insurance. Eventually, also at least one medicine applied in the treatment should be covered. Listings 6.2 to 6.6 show the corresponding queries and Listing 6.1 the definition of the used namespace prefixes. The queries were aligned to the used dataset (see Section 6.2.2) in the sense, that an evaluation of each query on a local RDF triple store with SPARQL endpoint, integrating all distributed data in Triple Space System, would yield a non-empty result.

```
PREFIX rdf:            <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:           <http://www.w3.org/2000/01/rdf-schema#>
PREFIX district:       <http://www.districts.org/>
PREFIX districts:      <http://www.districts.org/districts#>
PREFIX addresses:      <http://www.districts.org/addresses#>
PREFIX medical:        <http://www.medicalcare.org/>
PREFIX medics:         <http://www.medicalcare.org/medics#>
PREFIX drugs:          <http://www.medicalcare.org/drugs#">
PREFIX treatments:     <http://www.medicalcare.org/treatments#>
PREFIX insurances:     <http://www.medicalcare.org/insurances
```

Listing 6.1: Common namespace prefixes for queries

```
CONSTRUCT { ?med medics:provides treatments:treatment_1341. }
WHERE { ?med medics:provides treatments:treatment_1341. }
```

Listing 6.2: Query 0

```
CONSTRUCT { ?med medics:locatedAt ?address .
   districts:district_1 districts:contains ?address .
   ?med medics:provides ?treatment . }
WHERE { ?med medics:locatedAt ?address .
   districts:district_1 districts:contains ?address .
   ?med medics:provides ?treatment . }
```

Listing 6.3: Query 1

```
CONSTRUCT { ?med medics:locatedAt ?address .
   districts:district_1 districts:contains ?address .
   ?med medics:provides ?treatment . }
WHERE { ?med medics:locatedAt ?address .
   districts:district_1 districts:contains ?address .
   ?med medics:provides ?treatment . }
```

Listing 6.4: Query 2

```
CONSTRUCT { ?med medics:locatedAt ?address .
   districts:district_1 districts:contains ?address .
   ?med medics:provides treatments:treatment_134 .
   ?med medics:accepts ?insurance .
   ?insurance insurances:covers_treatment treatments:treatment_134 . }
WHERE { ?med medics:locatedAt ?address .
   districts:district_1 districts:contains ?address .
   ?med medics:provides treatments:treatment_134 .
   ?med medics:accepts ?insurance .
   ?insurance insurances:covers_treatment treatments:treatment_134 . }
```

Listing 6.5: Query 3

```
CONSTRUCT { ?med medics:locatedAt ?address .
   districts:district_1 districts:contains ?address .
   ?med medics:provides treatments:treatment_19252 .
   ?med medics:accepts ?insurance .
   ?insurance insurances:covers_treatment treatments:treatment_19252.
   treatments:treament_19252 treatments:suggests ?drug.
   ?insurance insurances:covers_drug ?drug . }
WHERE { ?med medics:locatedAt ?address .
   districts:district_1 districts:contains ?address .
   ?med medics:provides treatments:treatment_19252.
   ?med medics:accepts ?insurance .
   ?insurance insurances:covers_treatment treatments:treatment_19252.
   treatments:treatment_19252 treatments:suggests ?drug.
   ?insurance insurances:covers_drug ?drug . }
```

Listing 6.6: Query 4

6.2.2 Data Set and Data Distribution

The operation dataset was created synthetically and holds around 1.4 million triples. Its logical schema is defined as follows:

Namespace See Listing 6.1

Classes and Properties

Class medical:medical
- rdfs:label (literal: String)
- rdf:type (rdfs:Class)
- medics:locatedAt (districts:address)
- medics:accepts (medical:treatment)
- medics:accepts (medical:insurance)

Class district:district
- rdfs:label (literal: String)
- rdf:type (rdfs:Class)
- districts:contains (districts:address)

Class district:address
- rdfs:label (literal: String)
- rdf:type (rdfs:Class)

Class medical:insurance
- rdfs:label (literal: String)
- rdf:type (rdfs:Class)
- insurances:covers_drug (medical:drug)
- insurances:covers_treatment (medical:treatment)

Class medical:drug
- rdfs:label (literal: String)
- rdf:type (rdfs:Class)

Class medical:treatment

- rdfs:label (literal: String)
- rdf:type (rdfs:Class)
- treatments:suggests (medical:drug)

For the test runs the dataset was partitioned and distributed to spaces and host machines as shown in Table 6.1. Thereto, these spaces are all root spaces.

Space	Host Machine	Subject Class	# Triple Instances
S1	M1	medical:medic	30310
S2	M1	medical:medic	30266
S3	M1	medical:medic	30266
S4	M2	district:address	100010
S5	M2	district:address	100010
S6	M2	district:address	100013
S7	M2	district:district	100010
S8	M2	district:district	100010
S9	M2	district:district	100013
S10	M3	medical:drug	63213
S11	M3	medical:drug	63415
S12	M3	medical:drug	63432
S13	M4	medical:insurance	99252
S14	M4	medical:insurance	99252
S15	M4	medical:insurance	99252
S16	M3	medical:treatment	28919
S17	M3	medical:treatment	28929
S18	M3	medical:treatment	28879

Table 6.1: Comparison of SPARQL query engines that provide query decomposition

6.2.3 Evaluation Results

We executed 10 test runs for each query with our proposed algorithm (see Chapter 4) and with a simplified flooding approach, respectively. Latter provides perfect recall by strong simplification of our algorithm regarding the preservation of reasonable query processing costs: During the stage of physical optimization and execution (see Section 4.5) at step 2(b)iv of the algorithm, instead of selecting a single data source space for the current subgraph's evaluation all data source spaces for the subgraph are chosen. All test queries were initiated on machine M1 (see Table 2 for its local spaces).

Figure 6.1 shows the average time for 10 successful runs for query 0 to 3. For query 0 the execution times of both the optimized and flooding algorithm variant are virtually identical, since in both cases no joins across several data source spaces taking place. Furthermore, both process query 2 and 3 faster than query 1 due to the fact, that, in comparison, the subqueries executed for query 1 select huge segments of the global dataset (approx. 100,000 addresses and 100,000 medics for flooding). The evaluation of query 4 needed for the optimized execution 215 seconds in average, the flooding execution was not able to produce any results after running 10 minutes at each test run.

In Table 6.2 we compare for our optimized algorithm the number of test runs, which yielded a non-empty answer, to the total number of runs. Except from query 4, all test runs terminated more often successfully than unsuccessfully, while consuming at most half of the time the flooding algorithm consumed for solving the same query. Furthermore, we present the number of returned solution bindings for both alternatives.

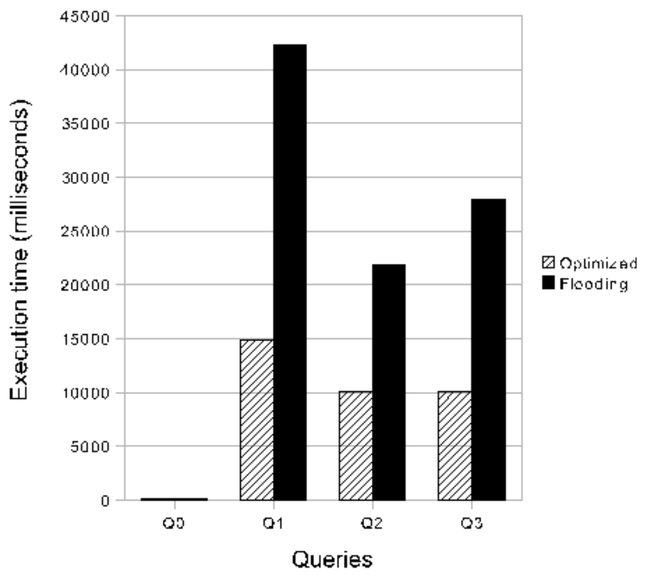


Figure 6.1: Query processing times

Query	# Successful Runs	# Overall Runs	# Solution Bindings (optimized, non-perfect recall)	#Solution Bindings (flooding, perfect recall)
Q0	10	10	1	1
Q1	10	10	127	6045
Q2	9	10	17	17
Q3	6	10	5	7
Q4	3	10	1	-

Table 6.2: Comparison of SPARQL query engines that provide query decomposition

As conclusion of our evaluation, we assert that for the problem of getting at least one solution our optimized approach, which cautiously selects a single data source for each subgraph, yields a tremendous performance gain in contrast to the naive strategy of asking every available data source space. In addition, it has been shown that the occurrence of unsuccessful test runs is relatively small compared to the gain of performance. Hence the QPP’s trade-off of perfect recall for better performance is well-balanced and highly recommendable. This outcome is even more remarkable considering the rather simple metadata made accessible.

7 Conclusion and Future Work

In this document we have presented the background for the Distributed Semantic Query Tool (DSQT) and the Query Preprocessor (QPP). These two components have been developed as software deliverables in the TripCom project. These components can be exploited by software developers who wish to develop applications with Triple Space technology.

DSQT has been developed to push the state-of-the-art in constructing SPARQL queries with special provisions to operate against the Triple Space System. DSQT provides the users with a graphical user interface which guides them through the process of constructing a new query and helps them understand the eventual query evaluation cost before being used in an application.

QPP offers the Triple Space System the means to facilitate *adaptive distributed query processing* oriented towards a time-efficient delivery of a small subset of all possible query answers retrievable by perfect recall. The QPP decomposes a SPARQL query into subqueries, distributes the subqueries to kernels for evaluation and, eventually, reassembles the answers for the subqueries to the overall result of the initial query. All actions are efficiently executed by alignment to logical properties of the query and physical characteristics of the distributed environment comprising the DM's triple pattern index and the database statistics for each space. During our evaluation test runs, we have been able to measure a tremendous improvement of the execution time compared to the strategy of 'asking each data source for a subquery'. Despite the non-perfect recall of our approach most of the tests returned non-empty results as desired.

A great potential of improving the QPP lies in the distinct analysis of the relationship between necessary recall percentage and costs in the terms of different types of evaluation scenarios, i.e. types of data populations. Moreover, it seems apparent that a more detailed form of metadata for the spaces' datasets can lead to an even bigger reduction of execution time while maintaining the same amount of successful query execution. This more sophisticated metadata could include further statistical RDF data, for instance the selectivities for subject and objects related to a distinct predicate. Apart from that a new kind of metadata the information on the machines' resources, .e.g. CPU/RAM load, running the kernels seems promising.

8 REFERENCES

[1] Bradner, S. IETF RFC 2119: Key words for use in RFCs to Indicate Requirement Levels, 1997.

[2] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73--170, 1993.

[3] Laura M. Haas and Donald Kossmann and Edward L. Wimmers and Jun Yang. Optimizing queries across diverse data sources. *In Proc. of VLDB*, pages 276--285, 1997.

[4] Donald Kossmann. The State of the art in distributed query processing. *ACM Comput. Surv*, 32(4):422--469, 2000.

[5] Andreas Langegger and Wolfram Wöß and Martin Blöchl. A Semantic Web Middleware for Virtual Data Integration on the Web. In Manfred Hauswirth and Manolis Koubarakis and Sean Bechhofer, editors, *Proceedings of the 5th European Semantic Web Conference* in LNCS, Berlin, Heidelberg, 2008. Springer Verlag.

[6] Eric Prud'hommeaux. Optimal RDF Access to Relational Databases. Technical report, 2004. http://www.w3.org/2004/04/30-RDF-RDB-access/.

[7] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, W3C, 2008. http://www.w3.org/TR/2008/ REC-rdf-sparql-query-20080115/.

[8] Bastian Quilitz and Ulf Leser. Querying Distributed RDF Data Sources with SPARQL. In Sean Bechhofer and Manfred Hauswirth and Jörg Hoffmann and Manolis Koubarakis, editors, *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings* in Lecture Notes in Computer Science, pages 524--538, 2008. Springer.

[9] Russell, A., Smart, P., Braines, D. and Shadbolt, N. (2008) NITELIGHT: A Graphical Tool for Semantic Query Construction. In: *Semantic Web User Interaction Workshop (SWUI 2008)*, 5th April 2008, Florence, Italy.

[10] Andy Seaborne. ARQ - A SPARQL Processor for Jena. 2008.

[11] Andy Seaborne. Jena – A Semantic Web Framework for Java. 2008.

[12] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38-49, 1992.

[13] Amit Singhal, Modern Information Retrieval: A Brief Overview. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 24 (4): 35—4, 2001.