



D4.3.1 Annex: Distributed Indexing and Querying of Distributed Knowledge

Nicholas Gibbins (University of Southampton)

Andrew Douglas (University of Southampton)

Abstract

EU-IST Specific targeted research project (STREP) IST-2004-026460 TAO
Deliverable D4.3.1 (WP 4)

In this deliverable we present the TAO Distributed Knowledge Store (DKS), a prototype scalable Semantic Web repository for the storage of RDF. First, we discuss the requirements that have informed the design of DKS, then the architecture and implementation decisions.

Keyword list: RDF, Semantic Web, SPARQL, triplestore, space-filling curves

WP4 Scalable, Heterogeneous Knowledge Stores Document ID: **TAO/2008/D4.3.1/v1.0**

Nature: **Report**

Dissemination: **PU**

Contractual date of delivery: **new deliverable, accompanying D4.3.1**

Actual date of delivery: **28 August 2008**

Web links: **<https://forge.ecs.soton.ac.uk/projects/tao/>**

TAO Consortium

This document is part of a research project partially funded by the IST Programme of the Commission of the European Communities as project number IST-2004-026460.

University of Sheffield

Department of Computer Science
Regent Court, 211 Portobello St.
Sheffield S1 4DP
UK
Tel: +44 114 222 1930
Fax: +44 114 222 1810
Contact person: Kalina Bontcheva
E-mail: K.Bontcheva@dcs.shef.ac.uk

Mondeca

3, cité Nollez
75018 Paris
France
Tel: +33 (0) 1 44 92 35 03
Fax: +33 (0) 1 44 92 02 59
Contact person: Jean Delahousse
E-mail: jean.delahousse@mondeca.com

University of Southampton

Southampton SO17 1BJ
UK
Tel: +44 23 8059 8343
Fax: +44 23 8059 2865
Contact person: Terry Payne
E-mail: trp@ecs.soton.ac.uk

Sirma Group Corp., Ontotext Lab

Office Express IT Centre, 5th Floor
135 Tsarigradsko Shosse Blvd.
Sofia 1784
Bulgaria
Tel: +359 2 9768 303
Fax: +359 2 9768 311
Contact person: Atanas Kiryakov
E-mail: naso@sirma.bg

Atos Origin Sociedad Anonima Espanola

Dept Research and Innovation
Atos Origin Spain, C/Albarracin, 25, 28037
Madrid
Spain
Tel: +34 91 214 8835
Fax: +34 91 754 3252
Contact person: Alberto Capellini
E-mail: alberto.capellini@atosresearch.eu

Dassault Aviation SA

DGT/DPR
78, quai Marcel Dassault
92552 Saint-Cloud
Cedex 300
France
Tel: +33 1 47 11 53 00
Fax: +33 1 47 11 53 65
Contact person: Farid Cerbah
E-mail: Farid.Cerbah@dassault-aviation.com

Jozef Stefan Institute

Department of Knowledge Technologies
Jamova 39
1000 Ljubljana
Slovenia
Tel: +386 1 477 3778
Fax: +386 1 477 3131
Contact person: Marko Grobelnik
E-mail: Marko.Grobelnik@ijs.si

Executive Summary

A key requirement of the TAO transitioning methodology and suite is the provision of scalable storage and query facilities for semantic data, for use by the TAO suite of software. In the current generation of Semantic Web applications, scalability is a key issue; these now commonly must be able to query knowledge bases (in RDF or OWL) in excess of 10E9 triples, and expect response times suitable for interactive applications. As a result, existing RDF triplestores are typically optimised for fast query responses, at the expense of import speed; for large applications, the bottleneck is frequently the time taken to update the contents of the knowledge bases, due to the cost of regenerating the indexes over the data.

This work focuses on techniques for building the next generation of RDF repositories, in particular on the use of distributed indexing and querying techniques for spreading the load across a cluster of machines. We have adopted a novel approach to RDF storage, using a space-filling (fractal) curve to both index the RDF data and to control the partitioning of the data across multiple servers. In this deliverable we present the initial version of the TAO Distributed Knowledge Store (DKS), a prototype scalable Semantic Web repository for the storage of RDF.

The release of DKS described in this report provides functionality for the storage, querying and retrieval of RDF data. This data can come from a variety of sources: semantic descriptions of web services, ontologies, and annotations of source documents, to name a few sources that are key to the TAO scenario. The DKS has been designed with standards compliance in mind; we use common Semantic Web technologies (RDF, SPARQL) both as the interface to the DKS, and as the internal communications framework between system components.

This software complements the federated Heterogeneous Knowledge Store (HKS) described in TAO Deliverable D4.2. Unlike the HKS, the components of the DKS are assumed to be homogeneous and under central control, and the DKS concentrates on scalable query only; inference is considered to be a separate consideration.

Contents

TAO Consortium	2
Executive Summary.....	3
Contents.....	4
1 Introduction	5
1.1 Relevance to TAO	5
1.1.1 Relevance to project objectives	5
1.1.2 Relation to other workpackages	5
1.2 Deliverable Outline.....	6
2 Design.....	7
2.1 Partitioning Scheme.....	7
2.2 System Architecture	11
2.2.1 Catalogue Server.....	11
2.2.2 Partition Servers	12
3 Installation Guide	13
3.1 Configuration.....	13
3.1.1 Partition Server Configuration.....	13
3.1.2 Catalog Server Configuration	14
3.2 Compilation	15
3.3 Installation	15
4 Conclusions and Future Work	16
5 References.....	17

1 Introduction

The transitioning of legacy applications to a more semantically-rich infrastructure is a knowledge-intensive task that uses a variety of data from different sources. This transitioning process is described by the TAO Methodology and supported by the tools in the TAO Suite, for whom a key point of coordination is the repository which maintains this data.

As with many knowledge-intensive tasks, the quantity of data used may be substantial, so there is a clear requirement for a scalable semantic repository that can store this data. This requirement is met by a combination of components. The Heterogeneous Knowledge Store (HKS) described in deliverable D4.2 provides scalable storage of both structured and unstructured data, and in this deliverable, we present the Distributed Knowledge Store (DKS), a scalable repository which concentrates on the storage and retrieval of structured RDF data[7].

1.1 Relevance to TAO

As a concrete set of tools for supporting the TAO Methodology, the TAO Suite requires facilities for storing, querying and manipulating the data used during the transitioning process. Of this data, a portion will be structured data with rich semantics, represented using the standard Semantic Web knowledge representation languages RDF and OWL.

Elsewhere in WP4, the Heterogeneous Knowledge Store is responsible for providing storage and retrieval facilities over a broad variety of data, including semantic data. The Distributed Knowledge Store complements the HKS by providing scalable storage for RDF data. Unlike the HKS, it concentrates on query alone, and does not consider issues of inference. In this section, we summarise the relevance of the DKS to the overall objectives of TAO, and to the other workpackages within TAO.

1.1.1 Relevance to project objectives

The Distributed Knowledge Store is related to the second and third project objectives of TAO (respectively, augmentation and integration of legacy content, and the transitioning methodology and infrastructure). In relation to the former objective, it provides storage facilities for the knowledge about the legacy application being transitioned, while in relation to the latter objective, it will be part of the TAO Suite (either used directly by the other components, or used indirectly as a storage backend for the HKS).

1.1.2 Relation to other workpackages

The Distributed Knowledge Store provides a general-purpose storage capability for Semantic Web (RDF) data. As such, it can be used to partially fulfil several of the requirements of the components of the TAO Suite, specifically the tools produced by in WP2 and WP3. For those situations where only Semantic Web data is to be stored (that is, where there is no additional textual content, for example, and where no inference is required), the DKS can be used by itself.

1.2 Deliverable Outline

This deliverable is structured as follows:

- **Section 2** describes the architecture of the system, and explains the multidimensional indexing techniques used.
- **Section 3** provides a guide to installing, configuring and running the DKS
- **Section 4** concludes the report and summarises the advanced functionality to be developed for D4.3.2.

2 Design

The design of the DKS has been informed by standard techniques for building distributed databases, and for implementing RDF storage. The common practice amongst existing RDF repository implementations is to treat a set of RDF statements as a relation with tuples that consist of the subject, predicate and object (and frequently also the graph) components of the statements [9][10][11]. An adaptation of this approach also underpins the translation of SPARQL[8] into the relational algebra [3]; in this translation, the triple patterns of which a SPARQL query is comprised are mapped onto a projection of a selection over a notional Triples relation. Although there have been recent developments using vertically partitioned decomposed storage models for RDF on a column-oriented DBMSes [12], in which each RDF property is stored in a separate table, the ‘triple table’ approach is still predominant.

2.1 Partitioning Scheme

We treat the distribution of the data in the DKS as a horizontal partitioning of this notional triple table across the servers in the system, in which triples are assigned to different servers for storage. These partition servers (PS) are considered to be black boxes for the purpose of the high-level system design; we assume that they provide a restricted interface that allows for storage and querying of data.

The nature of this horizontal partition is our main concern in the design of the DKS; a good partitioning scheme will evenly spread both the storage cost and the query processing cost across the servers in the system, and will improve query response times. While it would be possible to choose a partitioning scheme that is based on a particular ontology, in which information about a particular class of resources or property is held on a certain server, such schemes make it harder to perform effective load-balancing; assumptions that were made about the distribution of data in a small dataset (for example, the number of instances of class X compared to class Y) may not hold for a larger dataset which uses the same ontology.

Although the efficiency of the query optimiser probably has the single greatest effect on the query performance of a database as a whole, it is still strongly affected by the choice of partitioning scheme. In a distributed database, the selectivity of the partitioning scheme is key to reducing the number of unnecessary queries of partitions. We have chosen to use a partitioning scheme which is independent of the ontology used, in order to facilitate a more even distribution of data. The literature contains a number of multi-dimensional indexing techniques which may be used to generate a suitable horizontal partition; [5] contains a summary of these techniques.

Our partitioning scheme is based on the use of fractal, space-filling curves such as the Hilbert or Peano curve, and follows work on multidimensional indexing using these curves [1][2][4]. To explain how these techniques work, it is useful to consider a geometric approach to the problem of partitioning the data in an RDF repository.

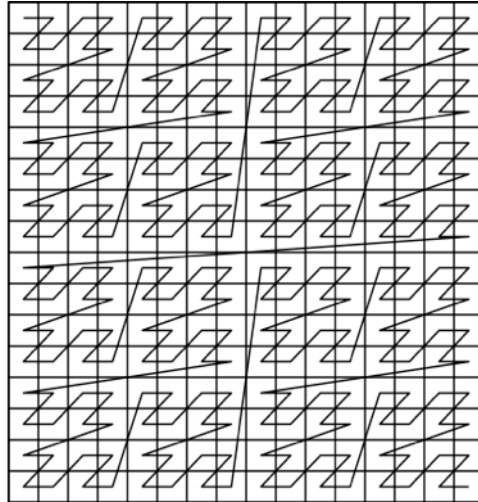
In addition to the triple table representation, we can also consider the triples in an RDF dataset to be a set of points that lie in a multidimensional space. The dimensions of this space correspond to the range of possible values for the subject, predicate and object of the triples; a triple `<ex:John> <rdf:type> <foaf:Person>` is a point located at

ex:John on the subject axis, rdf:type on the predicate axis, and foaf:Person on the object axis. We can therefore view a partitioning of the triples as a partitioning of the space in which the triples are located; a good partitioning would be one in which each partition contains roughly the same number of points.

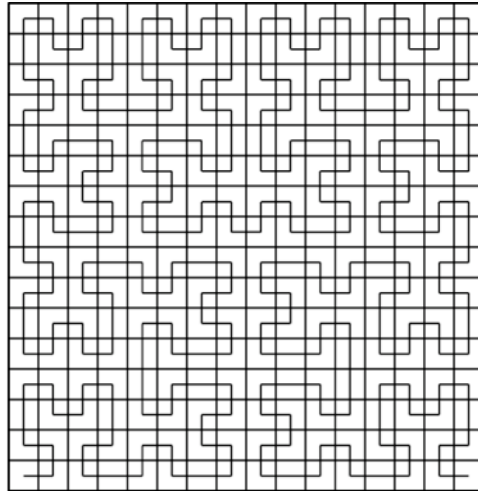
We can extend this triple space point of view to consider queries. A SPARQL query pattern `?s <rdf:type> <foaf:Person>` corresponds to a line running through the space; triples that satisfy this pattern correspond to points which lie on this line. Similarly, a query pattern with two variables (for example, `<ex:John> ?p ?o`) corresponds to a plane in the space. Given a set of volumes in the triplespace which correspond to partitions of the set of triples, we can determine whether a given partition might contain a triple that satisfies some triple pattern by determining whether the volume corresponding to the partition intersects with the plane that corresponds to the triple pattern.

To ensure that the points are evenly distributed in the space, and to make the dimensions of the space uniform, we can hash the URIs and literal values that make up the triples; a good hash function will spread the data evenly through the triple space. However, some dimensions are less even than others by virtue of being more sparsely populated. For RDF data, there are typically fewer predicates in use in a given dataset than there are distinct graphs, and fewer graphs than there are distinct subjects. The object dimension is the most populous because it contains both URIs and literal values. For this release, we use a 63 bit fragment of an MD5 hash of the values (URIs or literals) with a 1 bit in the LSB position to indicate whether the value is a URI or a literal.

A space-filling curve is a function which maps all of the points in a space of two or more dimensions to a one-dimensional line; it behaves like a line which passes through every point in the space exactly once. From a database perspective, a space-filling curve allows us to reduce the dimension of a dataset so that we can use existing indexing techniques that are designed for unidimensional data, such as the B-Tree. There are numerous kinds of space-filling curves, with varying properties: Hilbert, Peano (also known as Morton or Z-order), and Gray, to name three common examples [13]. These curves have been investigated with respect to their use for indexing multidimensional data in several places, most notably in [14][1][4], but have not been applied to the issue of RDF storage.



Peano (Z-order) Curve



Hilbert Curve

The approach that we have adopted for DKS uses the UB-Tree [4][6], which divides a multidimensional space into regions based on segments of a Z-order curve which covers the space, and then indexes the curve segments using a B+Tree. This is an attractive choice for two reasons, particularly when compared to the Hilbert curve based approach of [1]:

- Given a point in a multidimensional space, the calculation of the position of the corresponding point on a Z-order curve is a cheap operation; the ordinal position on the curve (the Z-index) can be calculated by performing a bit-interleaving of the binary representations of the coordinates on each dimension, as given by the equation below ($x_{j,i}$ is the i^{th} bit of the coordinate in the j^{th} dimension, d is the number of dimensions, and s is the number of bits in the coordinates for each dimension). The inverse of this operation, moving back from a Z-index to the spatial coordinates is similarly straightforward.

$$Z(x) = \sum_{i=0}^{s-1} \sum_{j=1}^d x_{j,i} 2^{i \cdot d + j - 1}$$

By comparison, the mapping to a point on the Hilbert curve is expensive, particularly as the number of dimensions grows. The method described in [15], as a representative approach, is significantly more expensive.

- Given a query plane, there is an efficient algorithm for determining which segments of a Z-order curve intersect the plane. This is key to the performance of a space-filling curve based indexing scheme. The algorithms for calculating intersecting regions mostly adopt the same general approach. Given a point on the curve, identify the next place at which the curve intersects with the query plane, and select the line segment which contains that point of intersection (this is an intersecting region). Taking the end of that line segment, iterate to find the next point of intersection, and repeat until the end of the curve is reached.

The efficiency of these algorithms very much depends on the complexity of finding the next point of intersection. For the UB-Tree algorithm described in [4], this process scales linearly with the number of dimensions; for the RDF storage use case, this is a constant time operation for all intents. The complexity of the overall algorithm therefore scales as the number of intersecting regions; if the data is evenly distributed throughout the space, the regions are balanced, and the regions are progressively subdivided to achieve some maximum number of data points per regions, this will be proportional to the total number of data points in the space (or in our case, RDF triples in the repository), so the algorithm as a whole scales linearly with the size of the dataset.

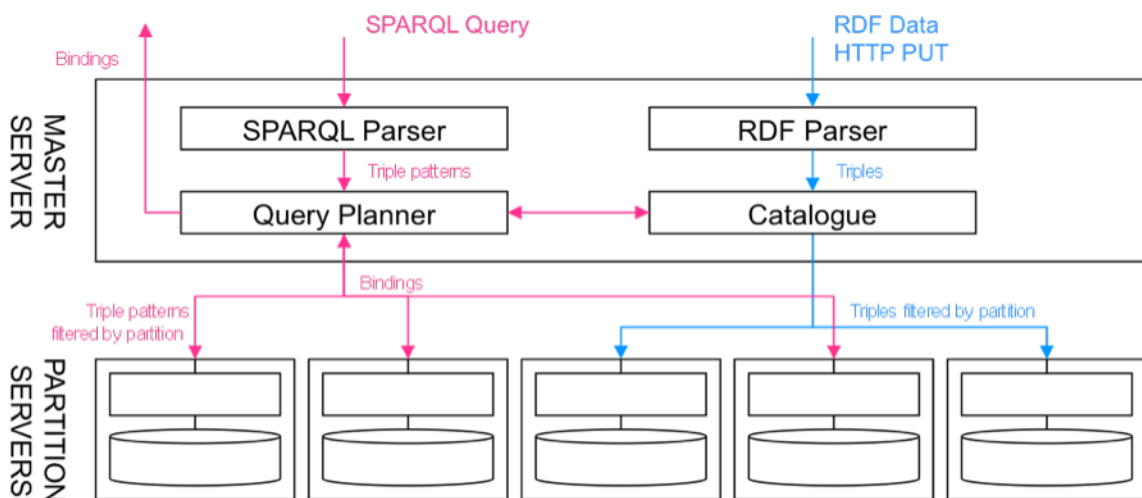
Although there are similar algorithms for the Hilbert curve, they are more complex. However, the Hilbert curve has the advantage that it is contiguous; adjacent points on the curve are adjacent in the space. By comparison, the Z-order curve features a number of discontinuities where it may jump to a distant point. For the RDF use case featuring hashed values, this is of lesser importance, because the majority of hashing functions do not preserve locality (that is, similar values do not have similar hashes).

The use of a single index structure for RDF data has considerable advantages over other common approaches. [16] describes the use of multiple indexes to cover the possible access patterns in an RDF query – for a query language that supports quads, there are sixteen possible access patterns (there are four positions in a quad, each of which may be either variable or ground), which require six indexes, assuming that prefix queries are possible. There is a considerable overhead in maintaining six parallel index structures; index rebuilding frequently appears to be the limiting factor when importing large quantities of RDF into current repositories. By reducing the number of indexes to one, the cost of maintaining the indexes is considerably reduced. Moreover, this advantage improves as the number of dimensions increases; if the approach described in [16] was used to index a repository which augmented quads with ORDI-style triplesets, the number of indexes required would rise to fifteen; the additional cost of supporting 5-tuples using UB-Trees is negligible, and would also have a negligible effect on the choice of hashing.

2.2 System Architecture

The architecture of the DKS, shown below, follows a conventional path for distributed databases, with a central query broker (the catalogue server) which is responsible for formulating and executing the query plan, and a cluster of partition servers which are responsible for storing the data.

The external interface to DKS uses standard Web technologies (HTTP-based model management, SPARQL queries). In addition, the interconnections between system components all use the same technologies wherever possible in order to maximise library reuse; for example, the communications between the catalogue server and the partition servers use a simplified profile of SPARQL over HTTP. The architecture of the system is summarised in the diagram below:



2.2.1 Catalogue Server

The catalogue or master server is the key component in the DKS. In theory, it is responsible for keeping track of the location of data within the system, in order to direct query components to the appropriate partition server. In practice, this means that it maintains a UB-Tree index which keeps track of the boundaries between the partitions, expressed as segment boundaries on the Z-order curve.

The catalogue server provides an external SPARQL endpoint for the system; on receipt of a query, it is responsible for decomposing the query into its constituent patterns, constructing a query plan, and executing that plan by contacting the relevant partition servers.

In addition, the catalogue server is used when RDF data is added to the system. When an RDF file is uploaded, the catalogue server uses the UB-Tree index to determine to which partition server each triple should be sent; for efficiency, the triples are collected into bundles and sent en masse, rather than being dispatched individually.

Finally, the catalogue server maintains additional metadata about the partitions and the loading of the partition servers in order to guide cost estimation for query optimisation, and the guide load balancing between partition servers. In this release,

the catalogue server maintains basic metadata for each partition (triple count and triple density) and histograms to help estimate the size of the returned relation for a given query pattern. However, this release does not yet contain a query optimiser, nor does it support load balancing through data migration between partition servers; these are both scheduled for the next deliverable.

2.2.2 Partition Servers

The partition servers are responsible for storing the RDF data in the system, and operate under the control of the catalogue server. Each partition server provides a basic query interface based on the HTTP binding for the SPARQL protocol, with a simplified profile of the SPARQL query language (limited to a single query pattern). In addition, each partition server provides a simple HTTP-based management interface; RDF data can be uploaded (by the catalogue server) using HTTP PUT, retrieved using HTTP GET and removed using HTTP DELETE.

In this release, the partition servers use the Jena library for their backend storage; the internals of the partition servers are not exposed to the catalogue server, in order to improve the loose coupling within the system.

3 Installation Guide

The TAO DKS is supplied as Java source. The current version can be downloaded from <https://forge.ecs.soton.ac.uk/projects/tao/>. The release of DKS described in this document is designed to work as an application running on a Java application server; it has been tested on the Glassfish application server¹, but will work on other Java EE 5 application servers. An installation of DKS therefore requires a preexisting installation of a suitable application server; you should follow the installation instructions for your chosen server.

3.1 Configuration

The components of the DKS are configured using the files in the `etc/` directory of the source distribution. These files are included in the application archives (`.jar/.ear/.war`) created by the build process; a change to the configuration of any component requires a fresh build of the software to include the new configuration files.

3.1.1 Partition Server Configuration

The partition server configuration file is located at `etc/store/server.xml`, and follows the format shown below:

```
<serverconfig>
  <tuplength>4</tuplength>
  <servername>Test Catalog</servername>
  <serverbasens>http://test.org/test</serverbasens>
  <db-url></db-url>
  <db-table></db-table>
  <db-user></db-user>
  <db-pass></db-pass>
</serverconfig>
```

The DKS can be configured for different tuple lengths using the **tuplength** directive: triples, for RDF without named graph support; quads, for RDF with named graph support; or quints, for RDF with named graphs and triplesets. By default, the partition servers are configured for quads.

The server is identified using the **servername** directive; this field contains a textual name for the server for debugging purposes.

The **serverbasens** directive contains the URI of the base namespace that is to be used for RDF sent to the server if no base namespace is defined in the RDF file.

The **db-url**, **db-table**, **db-user** and **db-pass** directives contain the database details required by Jena, which is used as a storage backend by the partition servers.

¹ <https://glassfish.dev.java.net/>

3.1.2 Catalog Server Configuration

The catalog server has two configuration files. The first, `etc/catalog/server.xml`, contains metadata and configuration information about the server itself, while the second, `etc/catalog/storeconfig.xml`, contains information about the partition servers which are to be controlled by the catalog server. In this release, the set of partition servers is a fixed configuration option which cannot be changed at runtime, but the ability to flexibly add partition servers to a running system will be implemented in a future release.

The `server.xml` configuration file strongly resembles the equivalent file for the partition servers, and follows the format shown below:

```
<serverconfig>
  <tuplength>3</tuplength>
  <servername>Test Catalog</servername>
  <savemethod>file</savemethod>
  <savefile>test.sv</savefile>
  <serverbasens>http://test.org/test</serverbasens>
</serverconfig>
```

The **tuplength**, **servername** and **serverbasens** directives all behave as they do for the partition server. The **savemethod** directive specifies the persistent storage for the UB-Tree index structure and partition server metadata, and takes the values **file** or **mysql**. If the value **file** is specified, the directive **savefile** gives the name of the file that will be used to store the flattened UB-Tree. Otherwise, if **mysql** is specified as the save method, the **db-url**, **db-table**, **db-user** and **db-pass** directives are used to specify a database which will be used to store the index and metadata.

The `storeconfig.xml` file describes the partition servers which are used by the catalogue server, and follows the format shown below:

```
<serverinformation>
  <server>
    <name>testserver</name>

    <url>http://localhost:8080/TripleStoreServer/TripleStoreServer</url>
    <protocol>SPARQL 1.0 (HTTP/1.1)</protocol>
    <owner>A.N.Other</owner>
    <description>A Test Server</description>
    <returntype>text/turtle</returntype>
    <returntype>application/rdf+xml</returntype>
  </server>
  ...
</serverinformation>
```

A storeconfig.xml file will contain a **server** block for each partition server in the system. Each partition server is identified with a URI (contained in the **url** element) which corresponds to the installation location of the servlet on the application server (see below). The protocol that is to be used to communicate with the server is specified by the **protocol** directive. For the current release, this defaults to “SPARQL 1.0 (HTTP/1.1)”; this directive is included to provide forward compatibility in the event that a revised SPARQL specification is published. The **returntype** directive is used to specify the Internet Media Types (MIME types) in which RDF may be returned by this partition server; the current supported types are RDF/XML (application/rdf+xml) and Turtle (text/turtle).

Finally, each partition server has some basic descriptive metadata (**name**, **description** and **owner**).

3.2 Compilation

The DKS distribution is supplied with an ant build file (build.xml) that will create the necessary files for the catalogue server and the partition servers. The catalog server can run either as a standalone application, or as a servlet running on an application server; consequently, there are targets that will build either, as follows:

ant catalog

build a .jar file for the catalogue server (a standalone application) copy it to build/jar/Catalog.jar

ant catalogserver

build .ear and .war files for the catalogue server (a servlet for use with an application server).

The partition servers in this release are always run as servlets. Correspondingly, there is a single build target for this component:

ant store

build .ear and .war files for the partition servers

The target **all** will build both the catalogue server and the partition server (defaulting to the standalone catalogue server), while the target **help** displays the available build targets.

3.3 Installation

To install the partition servlets or the catalogue servlet, follow the instructions for your Java EE5 application server; for the Glassfish server, this can be done either by using the autodeploy mechanism, or through the admin console.

To run the standalone catalogue server, type:

```
java -jar Catalog.jar
```

4 Conclusions and Future Work

The release of DKS described in this document provides a basic distributed RDF storage and query system. As it stands, this initial release does not compare favourably with existing RDF repositories; rather, this release is intended as a proof-of-concept which will be further developed in deliverable D4.3.2. The list of proposed enhancements for the next release is as follows.

- Data Migration

In this release, partitions may be subdivided, but there is no provision for transferring partitions between partition servers as a means of load balancing. This load balancing must make due consideration of both storage and CPU costs (data volume versus query frequency). It is our intent that this load balancing be centrally coordinated through the catalogue server; while a peer-to-peer approach would bear further investigation, we consider it beyond the scope of this project.

- Enhanced Query Planner

The current query planner is naïve, and attempts to retrieve all the intermediate relations from the partition servers between performing any joins. As a result the intermediate relations are frequently large, because the system does not order operations to make use of known variable bindings. We therefore plan to implement an enhanced query planner which uses the metadata collected and maintained by the catalogue server to inform a cost estimation-based query optimisation (by reordering the join operators in the query tree to reduce the size of the intermediate relations).

Once these enhancements are in place, we will perform a thorough comparative performance evaluation of the DKS against other RDF repositories. Finally, we intend to continue the ongoing technical cooperation with Ontotext (Sirma Group) to ensure that the DKS can be used as a scalable storage backend for the Heterogeneous Knowledge Store produced in D4.2.

5 References

- [1] J.K. Lawder and P.J.H. King. (2000) Using Space-Filling Curves for Multi-Dimensional Indexing. *Advances in Databases: Proceedings of the 17th British National Conference on Databases (BNCOD 17)*, Lecture Notes in Computer Science 1832, 20-35.
- [2] J.K. Lawder and P.J.H King. (2001) Querying Multi-dimensional Data Indexed Using the Hilbert Space-Filling Curve, *ACM SIGMOD Record* 30(1), 19-24.
- [3] R. Cyganiak. (2005) A relational algebra for SPARQL. HP Laboratories Technical Report HPL-2005-170.
- [4] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt and R. Bayer. (2000) Integrating the UB-Tree into a Database System Kernel. *Proceedings of the 26th International Conference on Very Large Databases*, 263-272.
- [5] V. Gaede and O. Gunther (1998) *Multidimensional Access Methods*, *ACM Computing Surveys*, 30(2), 170-231.
- [6] T. Skopal, M. Kratky, J. Pokorny and V. Snasel (2006) A new range query algorithm for Universal B-Trees, *Information Systems*, 31, 489-511.
- [7] G. Klyne, J.J. Carroll and B. McBride (2004) *Resource Description Framework (RDF): concepts and abstract syntax*, World Wide Web Consortium. <http://www.w3.org/TR/rdf-concepts/> (accessed August 2008)
- [8] E. Prud'hommeaux and A. Seaborne (2008) *SPARQL Query Language for RDF*, World Wide Web Consortium. <http://www.w3.org/TR/rdf-sparql-query/> (accessed August 2008).
- [9] S. Harris and N. Gibbins (2003) 3store: Efficient Bulk RDF Storage, *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems*
- [10] D. Beckett and J. Grant (2003) Mapping Semantic Web data with RDBMSes, *SWAD-Europe (IST-2001-34732) Deliverable 10.2* http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report (accessed August 2008)
- [11] R. Cyganiak (2005) Note on database layouts for SPARQL datastores, HP Laboratories Technical Report HPL-2005-171.
- [12] D.J. Abadi, A. Marcus, S.R.Madden and K. Hollenbach (2007) Scalable Semantic Web Data Management Using Vertical Partitioning, *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB2007)*, 411-422.
- [13] M. Mokbel, A. Arasu and I. Kamel (2003) Analysis of Multi-Dimensional Space-Filling Curves, *Geoinformatica*, 7(3), 179-209.

- [14] C. Faloutsos and S. Roseman (1989) Fractals for Secondary Key Retrieval, Proceedings of the Eighth Symposium on Principles of Database Systems, 247-252.
- [15] J.K. Lawder and P.J.H. King (2001) Using state diagrams for Hilbert Curve mappings, International Journal of Computer Mathematics, 78(3), 327-342.
- [16] A. Harth and S. Decker (2005) Optimized index structures for querying RDF data from the Web, Proceedings of the Third Latin American Web Congress (LA-WEB'05), 71-80.