Ingestion Pipeline for RDF

Nipun Bhatia, Andy Seaborne
Digital Media Systems Laboratory
HP Laboratories Bristol
HPL-2007-110
July 10, 2007*

ingestion pipeline, validation of RDF, inferencing, large RDF datasets

In this report we present the design and implementation of an ingestion pipeline for RDF Datasets. Our definition of ingestion subsumes: validation and inferencing. The design proposed performs these tasks without loading the data in-memory. There are several reasoners and Lint like validators available for RDF, but they require the data to be present in-memory. This makes them infeasible to be used for large data-sets(~10 Million triples). Our approach enables us to process large data-sets. The pipeline validates data-specific information constraints by making certain closed world assumptions and provides elementary inferencing support.

We illustrate the system by processing large data sets (~10 Million triples) from the Lehigh University BenchMark. We highlight the errors the system is capable of handling by writing our own ontology for an educational institute and data with errors in it.

* Internal Accession Date Only
Approved for External Publication

Ingestion Pipeline for RDF

Nipun Bhatia[1], Andy Seaborne[2]
nipun@ug.iiita.ac.in, andy.seaborne@hp.com

June 19, 2007

[1]Senior Undergraduate–Indian Institute of Information Technology, Allahabad
[2]HP Labs, Bristol

Abstract

In this report we present the design and implementation of an ingestion pipeline for RDF Datasets. Our definition of ingestion subsumes: validation and inferencing. The design proposed performs these tasks without loading the data in-memory. There are several reasoners and Lint like validators available for RDF, but they require the data to be present in-memory. This makes them in-feasible to be used for large data-sets($\sim$10 Million triples). Our approach enables us to process large data-sets. The pipeline validates data-specific information constraints by making certain closed world assumptions and provides elementary inferencing support.
We illustrate the system by processing large data sets($\sim$10 Million triples) from the Lehigh University BenchMark. We highlight the errors the system is capable of handling by writing our own ontology for an educational institute and data with errors in it.

Contents

Chapter 1

Introduction

> *The Semantic Web will bring structure to the meaningful content of web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users. Such an agent coming to clinic's Web page will know not just that the page has keywords such as "treatment, medicine, physical, therapy" (as might be encoded today) but also that Dr. Hartman works at this clinic on Mondays, Wednesdays and Fridays and that the script takes a date range in* yyyy-mm-dd *format and returns appointment times.*

The above extract from a *Scientific American* article by Tim Berners-Lee succinctly explains the vision of Semantic Web. The Semantic Web envisions a web that enables data interoperability across applications and organizations.

1.1 Semantic Web

The Semantic Web is a web of data[10]. It is a technology for sharing data and models, in a way that the Web is for human readable documents. It recommends use of certain standards(eg. RDF, OWL, RDFS) for data exchange, re-use and for vocabularies to describe that data. These technologies enable data integration and inference over existing data, which most times is intuitive to humans. Merging and sharing of data also enables execution of richer queries. The queries can access not only the specified data store but also data from other sources. URIs[9] play a fundamental role in all this. Theoretically, any person could publish RDF which uses the same URIs as your data and thus be linked to it. However, assuming consistency of data written by disparate sources is not an assumption within reasonable bounds.

The use of ontologies, designed in OWL or RDFS allows us to define concepts and relationships in data about a particular domain in the Semantic Web. The use of such technologies allows additional facts to be inferred which weren't necessarily present in the original RDF data.

The lower layers in the stack—URIs—in Figure 1.1[1] are the 'enabling technologies', the middle layers are the core 'RDF technologies' i.e. RDF—A standard syntax for

[1]The Semantic Web stack diagram(http://www.w3.org/DesignIssues/diagrams/sweb-stack/2006a.png) by Tim Berners-Lee

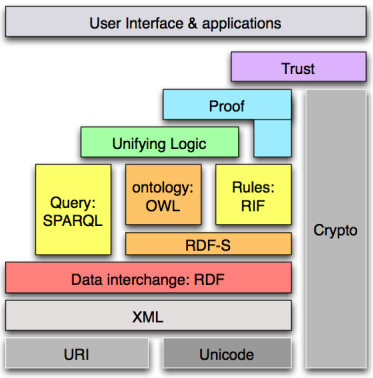

Figure 1.1: The Semantic Web Layer Cake

describing data and RDFS/OWL—for describing the relationships between resources, the upper layers i.e. logic(inferencing), proof and trust are open relevant research issues.

1.2 RDF

RDF[2] is one of the core technologies used to realize the Semantic Web. RDF is to Semantic Web what HTML was to the web, when it was in its infancy. In the current web, data is hidden away in HTML files, which is fit for human consumption but not for machines. Expressing data in machine processable common standards like RDF enables meaningful of aggregation, re-use and exchange of data from various sources. Disparate software applications can use data in RDF which was not written for or by them.

Unlike the tree structure of XML, RDF is centered around a flatter triple-based pattern[2]. Every XML document can have its own different schema and thus cannot be used for ad-hoc data integration as envisaged by the Semantic Web. In RDF the underlying model is—a directed labeled graph formed from triples—which depicts the subject, predicate and object. It has various serializations like RDF/XML, N-TRIPLES, TURTLE which are just mechanical things. The underlying model always remains the same.

1.3 RDFS and OWL

RDF Schema[3] provides vocabulary for describing properties and classes of RDF resources, with a semantics for describing hierarchies of such properties and classes. It enables designers to meaningfully add description of classes and properties.
The Web Ontology Language(OWL) has more facilities for expressing meaning and semantics than RDFS. It adds more relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry). It has three increasingly-expressive sub languages: OWL Lite, OWL DL, and OWL Full.

[2]A graphic depiction of the differences between RDF and XML can be found at http://www.semanview.com/c/RDFvsXML.html

RDFS and OWL though primilarily ontological languages used for inferencing can also be used for validation purposes. Inference enables entailments to be added to data based on connections expressed by schema, but not hard coded in data. The schema could be used to express that a particular property has a restriction that it should appear once(owlCardinality:1), but in the data the property doesn't appear. The reasoner, when inferencing over the data and schema, could add an entailment that ensures that the mentioned property appears once. This is perfectly valid as in the open world there could be somewhere where this property exists and the restriction holds. However, we could make a close world assumption and decide that since property doesnt appear in the data, the data is inconsistent with respect to this schema. This is the concept that we further explore in our report.

Chapter 2

Problem Description

2.1 Problem Statement

Design and implement an ingestion pipeline for RDF Dataset. The main aims of the pipeline are:

- Validation
- Inferencing
- Perform the validation and inferencing in-stream i.e. without loading the data into memory.

Validation, in the context of our system means raising a flag for inconsistencies in the data with respect to a schema. Inferencing by definition means to add entailments based on connections and concepts described in schema. It adds knowledge to the data by exposing implicit relations to the user. Our system aims to provide inferencing support without loading the data in-memory.

2.2 Motivation

The inability of the author of RDF data to immediately see the results of his work[6] is highlighted as one of the plausible reasons for the relatively slow growth of RDF data on the web. The inherent complexity in writing RDF which increases the probability of inadvertent errors further adds to the slow growth. An HTML author could add immediate value to his document by adding links to resources in his HTML data, URIs enable the same in Semantic Web. However, due to their length and structure it is not uncommon to make errors while writing them. Since the author is unable to get immediate feed-back on the RDF written by him, it may turn out that because of errors in URIs, the entire RDF is unable to link to other RDF resources. The errors in URIs are potentially fatal as an application using such data would be unable to link to other RDF resources and be unsuccessful in the harnessing the potential of linked RDF data.

In[7] Dave Reynolds et. al. propose a Semantic Integration portal based on data aggregation from various RDF sources. This aggregation is based on the fact that these sources publish RDF data about the same 'thing'. However, a missing declaration of a

resource as a class or mis-spelling in the name of a class can reduce a source of potentially useful data(by being about the same 'thing') to being untapped and unexplored.

The use of languages like RDF[2] and OWL[1] allow additional assertions to be added which are inferred from the class description and hierarchy. This reasoning ability is a vital facet of Semantic Web since it exposes hidden knowledge. There are a number of reasoners available that perform extensive in-memory reasoning for eg. Jena Inference Engine, Pellet. However, performing in-memory reasoning over large($\sim$10 million triples) datasets is computationally very expensive and can't be performed without specialized hardware. Some inference engines allow the data(A-box) and the ontology(T-box) specifying the links between the concepts, to be kept separately. Even in these reasoners the A-box(data) needs to be brought in-memory for the inferencing to be performed. Our system also makes the distinction between A-box and T-box, however instead of loading the contents of A-box into the memory for inferencing, we perform inferencing in-stream. Inferencing over a stream of RDF data, without storing all the triples in memory is computationally cheap and doesnt require specialized hardware. Though, the trade-off is that the inferencing is not as extensive as the one done in-memory.

The project aims to overcome the above mentioned issues. We hope to provide an ingestion pipeline that helps a publisher to weed out potential errors in his RDF data real time. It will also be useful for developers of Semantic Web application to check sources for errors. Through these initiatives, we take a small step in solving the larger problem of trust(Figure 1.1) in Semantic Web. We also hope to unearth implicit relationships between resources by developing a module that performs inferencing without loading the triples in-memory. Further, we aim to couple the processed data with a generic RDF data browser to enable a user to immediately see and browse over the RDF written.

2.3 Problem Description

As explained in the previous section, the ingestion pipeline tool-set proposes a solution to perform validation based on RDFS/OWL schema(ontology) and elementary inferencing over large sets of N-triples. To overcome the problem in processing of large set of N-triples we develop a system to perform the above tasks in-stream and not storing all the triples in-memory. The validation which makes a closed world assumption would subsume *URI Inspection, Literal Inspection, Property Inspection, Class Inspection and Cardinality Inspection.* The inferencing aims to perform a transitive closure over class-subclass and property-subproperty hierarchy. We also plan to provide support for inferencing over domain-range concepts.
The diagram of the system Figure 2.1 represents how the functionality of inspection and inferencing are plugged into the ingestion pipeline. A detailed diagrammatic representation of the system is presented in Figure 4.1. We further explain the concepts of validation and inferencing in the context of our pipeline in Chapter 4.

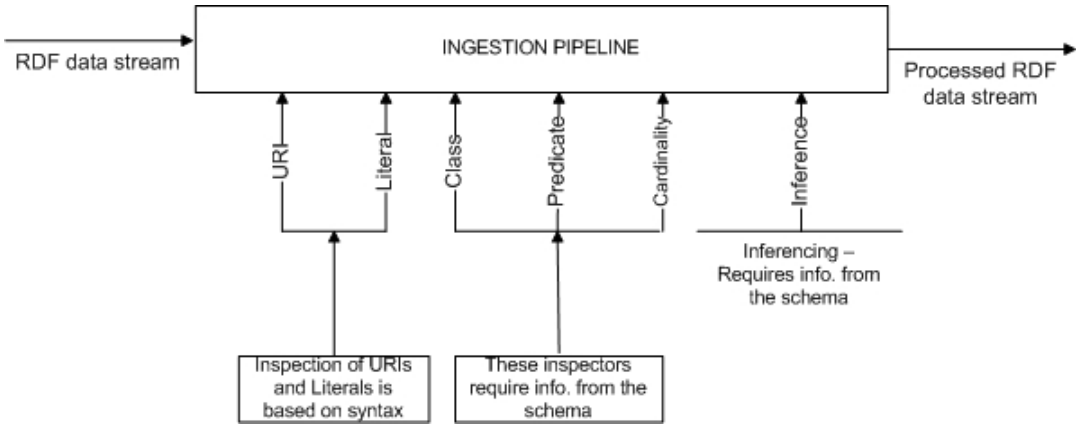

Figure 2.1: Overview of the System

Chapter 3

Related Work

3.1 RDF Validation

The term *validation* is not precisely defined with respect to RDF. It can be interpreted to mean a number of things. It can extend from a simple RDF syntax validation as done by the W3C validator[14], to validation based on RDFS/OWL schema(ontology) which makes a closed world assumption. The latter type of validation could be considered an RDF equivalent to what Lint does for C. Validators such as the W3C validator[14] don't perform these tasks. The central theme in them is to check the RDF for conformity with respect to the RDF Syntax.

3.1.1 Eyeball

Eyeball[5] is a standalone Jena based application that can be used for checking RDF(OWL) models for common problems. It is in-development and can currently be used as a command line tool or through a GUI. Eyeball is different from the W3C Validator in way that it checks RDF against user-defined schemas by making closed world assumptions. It checks RDF for problems that make it implausible but technically correct.
Eyeball can currently check[1] for:

- unknown [with respect to the schemas] predicates
- bad prefix namespaces
- ill-formed URIs [including in datatypes]
- ill-formed language tags on literals
- unexpected local names in schema namespaces
- untyped resources [including literals]
- individuals having consistent types assuming complete typing
- resources used as classes [objects of rdf:type, rdfs:domain, or rdfs:range; objects or subjects of rdfs:subClassOf] but not declared to be classes[i.e., not being of rdf:type rdfs:Class].

[1] For explanation of the error report generated, visit: http://jena.sourceforge.net/Eyeball/brief.html

- subjects that have a different number of values for some property than you'd expect from their cardinality-restricted types.

3.1.2 OWL Tidy

OWL Tidy[4] is a OWL Syntax Checker. It is a separately downloadable component of Jena2. It performs checks to report causes of syntactic non-conformance in OWL. It implements OWL Syntax checker as defined in the OWL Recommendation[2]. It supports checking in-stream and by storing the complete model in-memory.

3.2 Inferencing

There are a number of reasoners like Pellet and Jena Rules Engine[15] available for use. The Jena rules engine provides an implementation from a simple transitive reasoner to a reasoner for RDFS and OWL. Any of the reasoners provided by Jena can be bound to a model and perform the inferencing in-memory. The subproperty and subclass lattices are eagerly computed and stored in a compact in-memory form using the transitive Reasoner. The set of OWL reasoners in Jena provide a useful but incomplete implementation of the OWL Lite subset of the OWL Full language.

3.3 CWM

CWM[18] is a general-purpose data processor for the semantic web. It designed to perform the following tasks.

- Parse and print RDF formats: XML RDF, Notation3, and NTriples
- Store triples in a queryable triples database
- Perform inferences as a forward chaining FOPL inference engine
- Perform built in functions such as comparing strings, retrieving resources, all using an extensible builtins suite

Further, to perform validation CWM can take N3 files containing the RDF and DAML schema and a set of rules as input. These rules basically filter out inconsistencies and assist with validation. However, unlike OWL-Tidy the validation can't be done in-stream.

3.4 RDF Browsers

3.4.1 Tabulator

Tabulator[6] is one possible form of a generic browser for linked data on the web that can work with Firefox and Camino.It attempts to demonstrate and utilize the power of linked RDF data with a user-friendly browser that is able to recognize and follow RDF links to other RDF resources based on the users exploration and analysis. The

[2]See the OWL Test Cases and the OWL Semantics and Abstract Syntax for the formal definitions.

Tabulator[3] displays the RDF in the form of tables and graphs, loading RDF data from a resource whenever a user clicks on it.

There are many other semantic web data browsers. Some of these, such as Palm-DAML[11], RDF Author[17] and IsaViz[13] are generic, but have a limited ability to display data in the form of tables and graphs. Others such as CS-Aktive[8] and mSpace provide a powerful browsing environment, but are not generic. These browsers are unable to automatically dereference web information by looking up URIs and thus do not support the browsing of linked data. None of them, including Tabulator, 'validate' the data from a RDF resource before loading it.

3.4.2 Minimal Deductive Systems for RDF

S. Muñoz et al. in their paper[16] provide the semantics of a simple fragment of RDFS and the proof that it encompasses the main features of RDFS. They provide a formal proof for the semantics and a deductive system for it. This is potentially useful for our pipeline as it uses a lesser number of conceptual constructs. It might be computationally less expensive to provide support in-stream inference support for such semantics, yet have the same expressiveness of RDFS.

[3]Tabulator can be downloaded from: http://www.w3.org/2005/ajar/tab

Chapter 4

Plan of Work

As already explained 'RDF Validation' is not a precisely defined term. RDF content can't be validated in the sense that XML can be with respect to XML Schema. This is primarily because ontological languages such as RDFS and OWL are more for inferring than validating. Dave Reynolds in the article[1]–'Using RDFS or OWL as a schema language for validating RDF'–elucidates the issues in classifying data as invalid on the semantic web. There is not really such a thing as 'invalid' data on the Semantic Web: there is only fairly consistent, and inconsistent data. The ingestion pipeline we propose, explained in details in the following sections, flags inconsistencies in the data passing through the pipeline based on information from the schema and by making a closed world assumption. These additional assumptions not warranted in a general case but useful for validation purposes. As the data passes through the pipeline we also perform inferencing on it, but without storing all the triples in-memory. The following sections describe the inspectors and the inferencing support provided.

4.1 System Design

In Figure 2.1 we provide an overview of the system. As depicted in the figure, the data enters the ingestion pipeline and passes through the inspectors and the inferencing engine. Initially, we developed the inspectors using one comprehensive view of the RDF data. We assumed that declaration and hierarchy of classes(and properties) was present in the same file as the RDF data i.e. there is no separate ontology describing the data. This approach, however made the process of inferencing computationally expensive. To perform any meaningful non trivial inferencing we needed to store a large number of triples in memory or process the data twice. Both these approaches are infeasible for large datasets($\sim$10 million triples).

In view of the above problems, we modified our approach to segregate the data. The RDF data(A-box) that needs to be validated is processed separately from the ontology(schema aka T-box). This was based on the assumption that the ontology is small enough to be loaded and processed in memory. It is reasonable to make this assumption in most cases, except for biological ontologies. In the later sections we explain how this makes inferencing in our system computationally less expensive. Figure 4.1 shows the complete evolved system. In the next section, we explain each of the inspectors and

[1] http://esw.w3.org/mt/esw/archives/2004_03.html

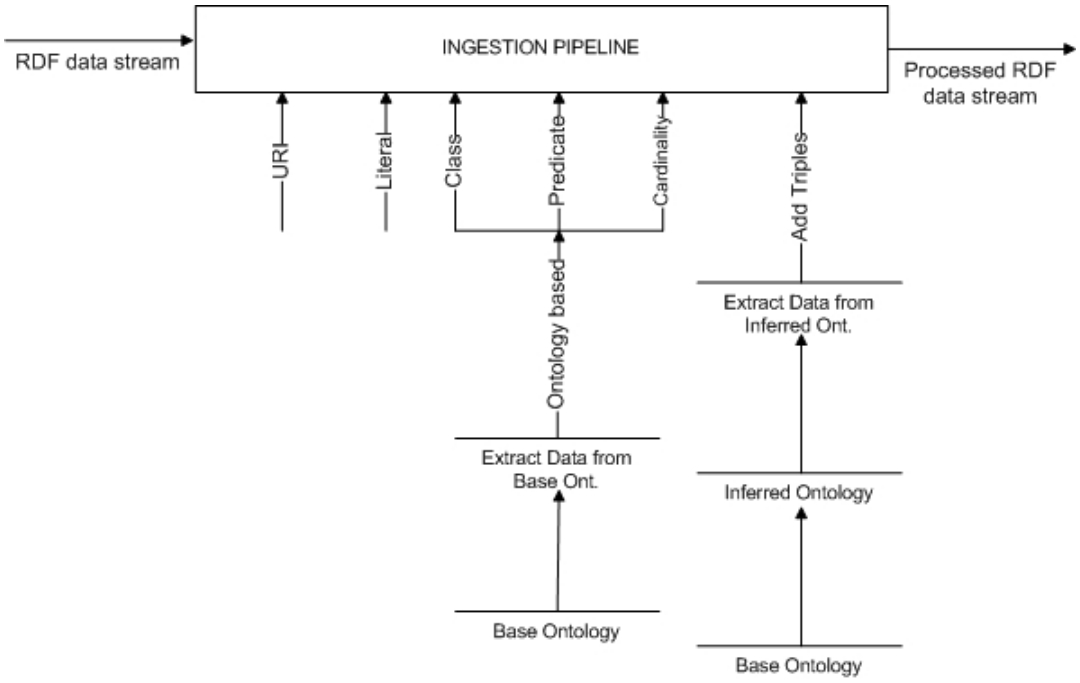

Figure 4.1: Detailed diagram of the System

their functions in detail. We follow this by explaining the methodology for adopted to perform inferencing.

4.2 System Description and Implementation

The main of our pipeline is to perform the validation and inferencing in-stream, without loading the triples in-memory. To implement this concept we use a null graph for input and intercept the add method on it. A model based on this null graph allows us to access the raw RDF data as nodes. It allows us to perform add but stores no triples. The triple to be validated and inferred over is intercepted by the add method of the null graph. The add method is overridden to perform validation and inferencing. Since the null graph stores no triples in-memory(It can't answer find) we are able to perform validation and inferencing on a large RDF data-set.
For the pipeline to perform its stated operations it is neccessary that the ontology is processed before the data to be validated passes through the pipeline. This is because the validation uses the set of *KnownClasses* and *KnownProperties* and the *CardinalityMap* which are formed only after the ontology is read. After the two sets and the CardinalityMap have been formed, the ontology is inferenced over by an in-memory reasoner. From the inferenced ontology, an InferenceMap is created which stores the information about the super-classes, super-properties and domain-range of properties. It is only after the creation of the sets(KnownClasses and KnownProperties), the CardinalityMap and the InferenceMap does the data pass through the ingestion pipeline. The creation and how resources are added to these data-structures is explained in greater detail in

the following sections.
As stated in Section 2.3 the validation process should subsume *URI Inspection, Literal Inspection, Property Inspection, Class Inspection and Cardinality Inspection.* The pipeline provides inference support to perform *transitive closure over classes* and *properties* and *inference over domain and range.* The following sections provide details of the inspectors and how the inferencing is performed.

4.2.1 Inspectors

As shown in Figure 4.1 the class, property and cardinality inspection are ontology based, while the URI and Literal inspection are based on syntax. Since the URI and Literal inspectors check the syntax of URIs and literals in a triple, they don't need any information from the schema. The class, property and cardinality inspectors require information about the declared classes, property and cardinality restriction. This information is extracted from the ontology when it is being loaded.

URI Inspector

The URI Inspector checks the data for errors in the URI. A broad reason for the system flagging a triple in the data for wrong URI is that the URI is syntactically illegal. The cases covered are:

1. URI contains spaces: Spaces are not legal in URIs. This usually arises from file URIs when the file has a space in its name. Spaces in URIs have to be encoded.
2. URI has unrecognised scheme: The system flags the URI if the scheme part is unknown. By default it knows the following four schemes: *http, ftp, file, and urn.* The error generally occurs from a typo. Currently, you can't tell the system about any other schemes.
3. URI has no scheme: The URI has no scheme at all. This usually happens when some relative URI hasn't been resolved properly, eg there's no xml base in an RDF/XML document.
4. Scheme should be lower-case: It flags of the triple in which scheme is not in lower case.

Literal Inspector

The Literal Inspector checks the literals with language codes(eg. de) to make sure that they conform to the general syntax for language codes.

Class Inspector

When the ontology is read by the system it extracts the resources declared as classes. These classes are stored in a set *KnownClasses*, which is kept in-memory. It considers the following triples as valid declaration of a class 'C':
C rdf:type rdfs:Class
C rdfs:subClassOf _
_ rdfs:subClassOf C

When the RDF data to be validated passes through the pipeline it checks that in all of

_ *rdf:type C*
_ *rdfs:domain C*
C rdfs:subClassOf D
C and D are present in the classes defined in the ontology which are stored in the set KnownClasses.

Property Inspector

While reading the ontology the system looks at the the statements it counts as declaring properties, automatically considering RDF and RDFS properties. Similar to classes, these properties are stored in a set, *KnownProperties* which is kept in-memory. A property 'P' is declared in any of the following ways:
P rdfs:domain _
P rdfs:range _
P rdf:type rdf:Property
_ *owl:onProperty P*

When the triple from the RDF data to validated passes through the pipeline, its predicate is checked to make sure it is declared. This is done by chekcing if it is present or not in the set of *KnownProperties* formed during the reading of the ontology.

Cardinality Inspector

The Cardinality inspector checks that your data has the right number of properties for resources of a certain type. When reading the ontology a CardinalityMap is generated which stores information of properties on which restrictions are applied, the kind of restrictions and their value. Using this cardinality map, the data is inspected for violations of the cardinality constraints.
Due to the design of our system, at any instant we process only one triple. However, to check for violation of cardinality constraints on a particular property, we need information of the number of triples with the same property and subject that have passed through the system. For these purposes we maintain a structure(in-memory) called *Triple Data Store*. The Triple Data Store maintains information, in a condensed form, of the triples that have passed through the system and whose predicate is present in the cardinality map. It keeps a count of the triples, with the same subject and predicate(which is also present in the cardinality map), that have already passed through the system. This count, along with the information of subject and predicate, decides if a triple has violated a constraint specified on its predicate.
Figure 4.2 shows the class diagram. Due to space constraints not all methods have been shown.

4.2.2 Inferencing

In this section we explain the inferencing performed on the data when passing through the ingestion pipeline. The current pipeline supports the following inferencing.

- Transitive closure over the class-subclass hierarchy
- Transitive closure over the property-subproperty hierarchy
- Inference over range and domain

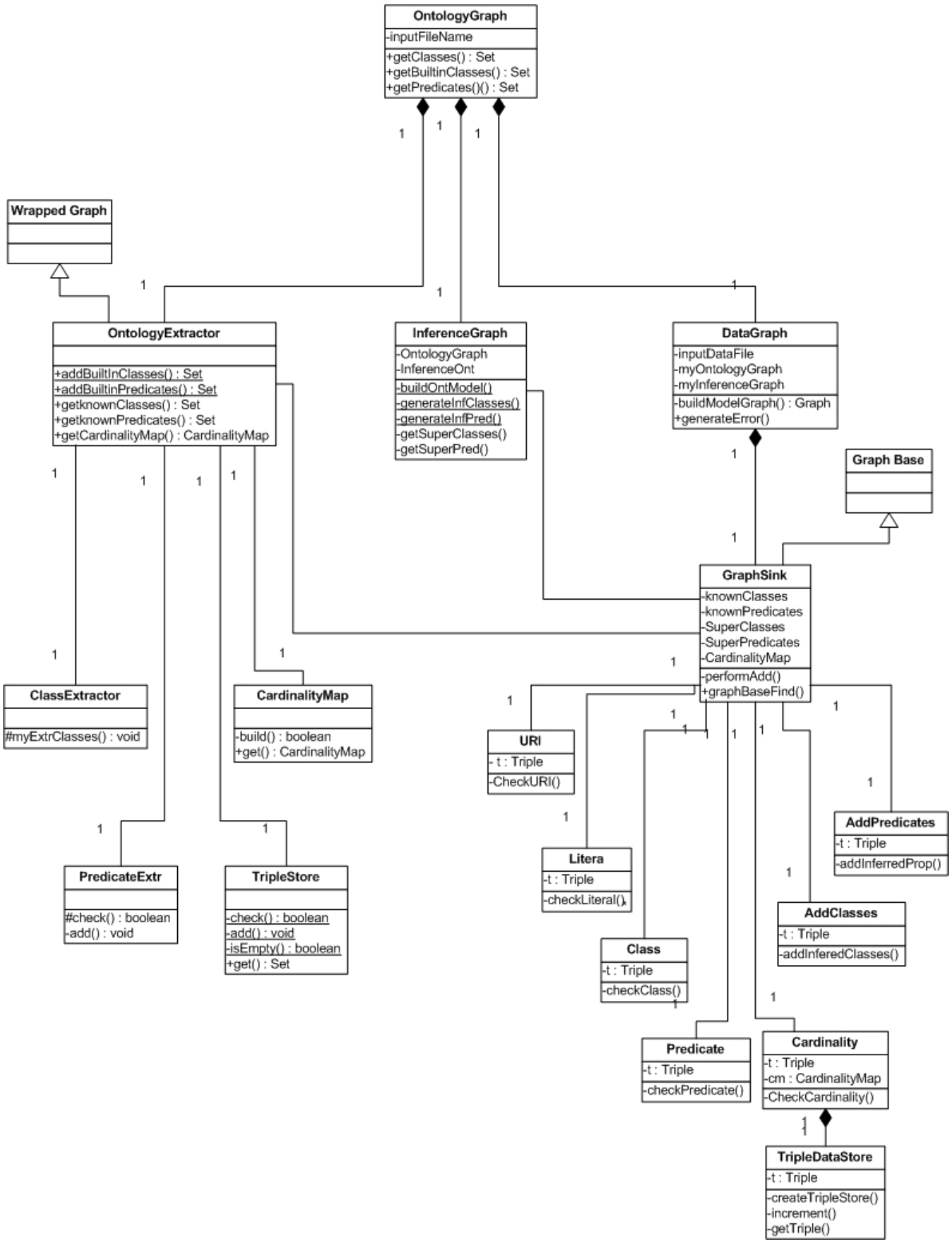

Figure 4.2: Class Diagram of the System

We explain the methodology adopted to perform the inferencing in the following paragraphs.

As already stated our pipeline requires that there is a strong separation between the data and schema(ontology). This condition helps us perform inferencing without storing all the triples in-memory.

Step 1: Load the schema

The schema is loaded in-memory and inferencing is performed on it using a standard Jena2 reasoner or Pellet reasoner. The added entailments and the schema are kept in-memory.

Step 2: Extract

From the above data we extract the super classes of the known classes[2], super properties, range and domain of the known properties[3]. We store this information(referred to as Inference Map) in a compact form in-memory.

Step 3: Inference

The Inference Map constructed above is used to perform the actual inferencing. The type of inferencing–transitive closure over classes, transitive closure over property or inference over domain-range–being performed depends on the 'kindof' triple being ingested.

Transitive closure over classes
The decision whether a triple is suitable for the transitive closure over classes to be performed on it or not, depends on the fulfillment of the condition that the triple being ingested has *RDF.type* as the predicate. It is then checked that whether or not the object is present in the known classes set. If it is not, no transitive closure over classes is performed. If it is, the list of super classes corresponding to that object is extracted from the inference map.
The algorithm below explains how we perform Transitive closure for classes. 't' is the triple over which we want to infer and 'tempList' is the list of super classes extracted from the inference map. As state above, the if condition is used to decide what 'kindof' triple it is.

$OC \leftarrow KnownClasses$ {Known Classes Extracted from Ontology}
if (*t.predicate* equals *RDF.type*) and (OC contains *t.object*) **then**
  $tempList \leftarrow List$ {List of super classes for t.object}
  while *tempList* hasNext **do**
    Add Class Entailments
  end while
end if

Transitive closure over properties
The transitive closure for properties is performed in an identical way as the transitive closure for classes. It is first decided whether the triple is suitable for transitive closure over property to be performed on it or not. This is decided on the basis of the

[2]This is the same set–*Known Classes*–that is formed for the class inspector
[3]This is the same set–*Known Properties*–that is formed for the property inspector

predicate. If the predicate is present in the set of known properties, it is suitable for transitive closure over properties to be performed over it. The list of super properties for this predicate is then extracted from the inference map.
The algorithm for it is similar to the one that performs transitive closure over classes.

Inferencing over domain and range
In addition to the elementary transitive closure over classes and properties explained above, we infer information from the domain and range of properties. If the schema specifies that a property *P rdfs:domain C* and the instance data has a triple of the type *x P y*, then system adds an entailment *x rdf:type C*. Similar, procedure is adopted to infer over the range of properties.
To perform this type of inferencing we first check whether the predicate is present in the set of known properties. If it is, the domain and range for that predicate is extracted from the inference map. Based on this information entailments are added to the data.

Step 4: Perform Step 3 for all triples

Step 3 is performed for each and every triple that passes through the ingestion pipeline.

It is important to note that the inferencing is being performed after the inspection. This is important as the during the inspection the set of known classes and known properties is formed which is used during the inferencing.

Chapter 5

Project Outcomes and Discussion

In this chapter we highlight the outcomes of the project. We have successfully proposed and implemented a pipeline that performs validation and inferencing without storing all the triples in memory. The exectuion times are directly proportional to the number of triples. To enable processing of a large set of RDF triples, the pipeline is developed to store a minimum number of triples. The only triples stored are the ones required for cardinality inspection i.e. triples whose predicates are present in the CardinalityMap. Even for these triples we dont store copies, but only maintain information in compact form. If the user can gurantee that the triples to be validated, are sorted on the basis of subjects, then no triples would be required to be stored.

In the following sections we illustrate the errors that the system can detect, explain the error messages generated and the display the inferencing results.

5.1 Example with Results & Discussion

5.1.1 LUBM

LUBM [12] or the Lehigh University Benchmark was developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way. It consists of a university domain ontology, customizable and repeatable synthetic data, a set of test queries, and several performance metrics. The data-generator provided generates synthetic OWL data over the Univ-Bench ontology in the unit of a university. Since the data is synthetic is doesn't have any errors in it. Inspite of this, it is a convenient source for generation of gigabytes of OWL data. To illustrate the capabilities of the system we manually introduced errors in the data.

Table 5.1 shows the results obtained after performing inferencing on the LUBM data using a standard Jena RDFS reasoner in its default configuration. The '-' in the table indicates that was not possible[1] to performing, as the data was too large to fit in the memory.

[1]On increasing the maximum heap size to 512MB, number of triples when input was 10 Universities were 1,805,997 & ratio 1.42

Universities	Originally	RDFS Reasoner	Ratio
1	100,543	144,793	1.44
3	477,784	679,515	1.42
6	884,143	1,255,497	1.42
10	1,272,575	-	-
50	6,657,562	-	-
80	10,734,220	-	-

Table 5.1: Inferencing with Standard RDFS reasoner

Table 5.2 shows the results obtained after inferencing on the LUBM data using the in-stream method proposed in the Section 4.2.2. Unlike the standard RDFS reasoner, the proposed system can perform inferencing over not only 1 Million triples (10 Universities), but also process greater than 10 Million triples(80 Universities). It is important to note that the ratio, in the both the tables, is stronlgy dependent on the data. For this given data set(LUBM University data) the ratios from the two methods are comparable. This however can't be considered to hold any and every data-set. The ratio obtained from an RDFS reasoner would be atleast *(*equal to)(best case) if not greater than the ratio obtained from our system. This is expected as the RDFS reasoner provides more extensive reasoning capabilities than the current version of our system.

Universities	Originally	In-Stream Reasoner	Ratio
1	100,543	138,664	1.38
3	477,784	653,027	1.36
6	884,143	1,206,510	1.36
10	1,272,575	1,735,612	1.36
50	6,657,562	9,082,154	1.36
80	10,734,220	14,641,154	1.36

Table 5.2: Inferencing in-stream

Though the data generated by LUBM is error-free, for illustration purposes we manually introduced a few errors in it. It is not possible to validate this data with Eyeball[5], as Eyeball also requires to keep the entire data in-memory. We have already seen, as in the case of inferring, this is not possible to do. Our system is able to detect these errors and raise the approriate flags. We removed the 'tt' from 'http:', the system was correctly able to detect that the resulting scheme 'hp' is unknown. It flagged off this triple with the URI error. As another error, we changed the spelling of one of defined classes. As excepted the system generated a class error and warned that the potential class is undeclared. We performed a similar test for predicates and literal.

5.1.2 Educational Institute

To illustrate the entire range of errors detected by the system we develop an ontology for an educational institute. We manually write data conforning with this ontology but introduce a few errors in it.

Figure 5.1 shows a graphical view of the ontology. Due to space constraints only some of the classes and properties are shown. All the labels of classes, URIs have been removed. The classes and properties shown are not of more importance than the ones

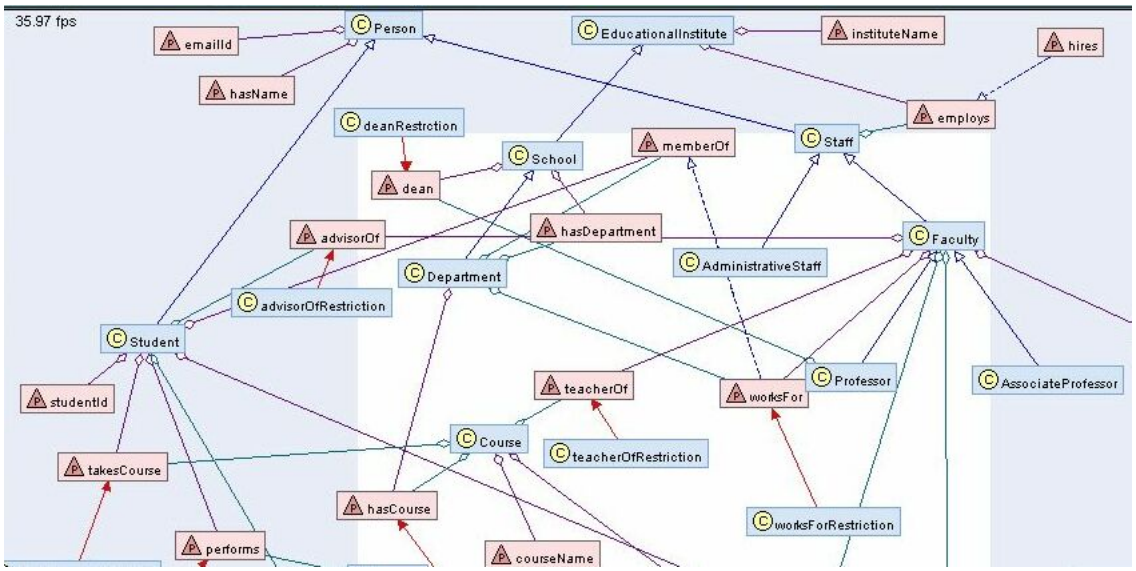

Figure 5.1: Grahpical view of the Ontology developed

ommitted. Since errors in the data were introduced keeping these classes and properties in mind, we have shown them for the reader to better relate to the example.

URI Error

The system on encountering a resource with malformed URI, generates a report of the kind:
URI ERROR! The URI: hp://soe.stanford.edu/ has unknown scheme.
This means that the system has detected an unkown scheme('hp://' in this case) in this URI.

Class Error

To flag off potential undeclared classes the system generates a statement in the report of the kind:

CLASS ERROR! Potential classes maybe undeclared: [http://localhost/educationalInstitute#University, http://localhost/educationalInstitute#SchOOl] - Check Ontology.

The resources enclosed between [ ] are undeclared as classes in the ontology(schema). As you would observe in Figure 5.1, in the ontology there is a class specified as 'School'.In the data a resource is specified to be type 'SchOOL'. Since there is no such class an error message to warn the user is generated. Similarly, in the ontology there is no class 'University'. However, in the data a resource is specified to be type 'University'. Since there is no such class, the class 'University' is added to the list of potentially undeclared classes.
A similar error report is generated for Predicates.

Cardinality Error

In the ontology developed a property 'dean' is specified on the class 'School'. A restriction is associated with this property. The restriction specifies that every school must have one dean(owlCardinality:1). The resctriction can be seen in Figure 5.1 in the top left hand corner. In the data this restriction is violated. The property for a particular subject has two values.
The error messages generated in this case is slightly more cryptic. The error message in the report looks something like:

CARDINALITY ERROR! Restriction type on property–educationalInstitute:dean with subject: http://med.stanford.edu is–owl:cardinality with value: 1, value in data is: 2.

In simpler language this means, the schema(ontology) specifies the restrcition owl:cardinality on the property– 'educationalInstitute:dean' with the value 1. However, in the data this property appears twice on the subject–'http://med.stanford.edu'. This is not consistent with the schema and hence the cardinality flag is raised.

Chapter 6

Conclusion

The system explained in the previous chapters highlights an approach to process large RDF datasets. We explain our assumption of having a strong sepration between the data and the schema(Ontology). We successfully illustrated our approach a with dataset of over 10 Million triples from the Lehigh University BenchMark. The system was also able to detect errors in the data that was written manually by the author. The data was written conforming to an ontology for an educational institute, but with a few errors. Along with the validation process we were able to perform inferencing on the dataset of over 10 Million triples. We were able to achieve this using the standard JVM[1] and no specialized hard-ware.

6.1 Limitations

The system proposed performs the validation and inferencing in-stream. However, as explained for the cardinality inspection we store a certain number of triples in the *triple data store*. For a very large dataset(∼50 Million), a case may arise in which the triple data store is large enough to not fit in memory. This problem is not unsolvable. In the triple data store, as explained, we store triples(in compact form) whose predicates is present in the cardinality map. It is not necessary to store these triples in-memory for the entire length of the execution. Certain triples can be removed from the triple data store after they have served their purpose. For example, there may be a certain property p in the ontology on which the restriction *minCardinality* with value *min* is specified. In the data stream, after observing a number of triples n with the subject s and property p, we can remove these triples from the data store as soon as $n > min$. It is important to note that in our current implementation of triple data store we dont maintain a copy of all the triples of a particular kind(eg. triples with subject s and property p). Our triple data stores information in a compact form which encompasses all the necessary information.

[1]The maximum heap size in a Standard JVM is 64MB

6.2 Restrictions & Future Work

Inspection

Though our system performs useful validation and inferencing, we note that a few more inspectors could be added to it. These inspectors on the lines of the ones present in Eyeball could increase the domain of errors detected by the system for eg. The URI inspector could be extended to enable the user to specify his own URI scheme.

User Options

Currently our system doesn't allow the user the option to switch off any particular inspector. This feature would be helpful in scenarios where the user doesn't feel the need to perform computationally and memory expensive inspection like Cardinality inspection.

Another area where an user-option would be helpful is in deciding which triples to totally supress. Currently the system allows suspect RDF to go through and doesn't supress them completely. This was partly motivated by the fact that there is no RDF that is wrong, but only inconsistent. The system, in current version, was designed only to flag inconsistent RDF. Future implementations could have an user-option to totally supress inconsistent RDF.

Input Types

The supported input type of RDF in our system is N-Triple. This can be severly restrictive for people publishing their data in other serilizations of RDF like RDF/XML or N3.To use the system in the current state they would have to convert their data to N-triples. It would be useful to provide support for these serializations of RDF.

Inferencing

The inferencing support is currently provided only for transistive closure of classes/properties and over domain and range. An OWL reasoner can provide much more extensive inferencing. The rules supported by an OWL reasoner could be studied in detail and the ones that are suitable to be implemented in-stream could be selected. The trade-off for their implementation would be reduced speed of computation and increase in the number of triples stored in-memory.

References

[1] S. Bechhofer. *OWL-Web Ontology Language*. http://www.w3.org/TR/owl-ref.

[2] D. Brickley. RDF-Resource Description Framework. http://www.w3.org/RDF, 2004.

[3] D. Brickley. RDF Schema. http://www.w3.org/TR/rdf-schema, 2004.

[4] Jeremy Caroll. *OWL–Tidy*. http://jena.sourceforge.net/contrib/contributions.html.

[5] Chris Dollin. *Eyeball*. http://jena.sourceforge.net/Eyeball.

[6] Berners-Lee et. al. Tabulator: Exploring and Analyzing linked data on the Semantic Web. *The 3rd International Semantic Web User Interaction Workshop, Athens, Georgia*, 2006.

[7] Dave Reynolds et. al. Semantic Information Portals. *World Wide Web Conference, Newyork, NY, USA*, 2004.

[8] H Glaser et. al. *CS AKTive Space: Building a Semantic Web Application*. Springer Verlag, 2004.

[9] I. Herman. *Introduction to the semantic web*. http://www.w3.org/2007/Talks.

[10] I. Herman. *The Semantic Web home page*. http://www.w3.org/2001/sw/, 2006.

[11] J. Margerison M. Dean. *PalmDAML*. http://www.daml.org/PalmDAML, 2002.

[12] Y. Guo Z. Pan and J. Heflin. An Evaluation of Knowledge Base Systems for Large OWL Datasets. *Third International Semantic Web Conference, Hiroshima, Japan, LNCS 3298*, 2004.

[13] E Pietriga. *IsaViz*. http://www.w3.org/2001/11/IsaViz, 2006.

[14] Eric Prud'hommeaux. *W3C Validator*. http://www.w3.org/RDF/Validator.

[15] Dave Reynolds. Jena 2 Inference. http://jena.sourceforge.net/inference, 2004.

[16] S. Muñoz J.Pérez. Minimal Deductive Systems for RDF. *4th European Semantic Web Conference, Innsbruck, Austria*, 2007.

[17] Damian Steer. *RDF Author*. http://rdfweb.org/people/damian/RDFAuthor, 2003.

[18] Dan Connolly Tim Berners-Lee. *CWM*. http://www.w3.org/2000/10/swap/doc/cwm.html, 2004.