Knowledge store version 1

Deliverable D6.2.1

Version Ready for Internal Review

Authors: Marco Amadori[1], Roldano Cattoni[1], Francesco Corcoglioniti[1], Bernardo Magnini[1], Michele Mostarda[1], Marco Rospocher[1], Luciano Serafini[1]

Affiliation: (1) FBK

Building structured event indexes of large volumes of financial and economic data for decision making

ICT 316404

Grant Agreement No.	316404
Project Acronym	NEWSREADER
Project Full Title	Building structured event indexes of large volumes of financial and economic data for decision making.
Funding Scheme	FP7-ICT-2011-8
Project Website	http://www.newsreader-project.eu/
Project Coordinator	Prof. dr. Piek T.J.M. Vossen VU University Amsterdam Tel. + 31 (0) 20 5986466 Fax. + 31 (0) 20 5986500 Email: piek.vossen@vu.nl
Document Number	Deliverable D6.2.1
Status & Version	Ready for Internal Review
Contractual Date of Delivery	December 2013
Actual Date of Delivery	December 18, 2013
Type	Prototype
Security (distribution level)	Public
Number of Pages	58
WP Contributing to the Deliverable	WP6
WP Responsible	FBK
EC Project Officer	Susan Fraser

Authors: Marco Amadori[1], Roldano Cattoni[1], Francesco Corcoglioniti[1], Bernardo Magnini[1], Michele Mostarda[1], Marco Rospocher[1], Luciano Serafini[1]

Keywords: knowledge store, unstructured content, mentions, entities

Abstract: Despite the widespread diffusion of structured data sources and the public acclaim of the Linked Open Data initiative, a preponderant amount of information remains nowadays available only in unstructured form, both on the Web and within organizations. While different in form, structured and unstructured contents speak about the very same entities of the world, their properties and relations; still, frameworks for their seamless integration are lacking. In this deliverable we present the first implemented version of the NewsReader KnowledgeStore, a scalable, fault-tolerant, and Semantic Web grounded storage system to jointly store, manage, retrieve, and semantically query, both structured and unstructured data. The KnowledgeStore plays a central role in the NewsReader project: it stores all contents that have to be processed and produced in order to extract knowledge from news, and it provides a shared data space through which NewsReader components cooperate. A description of the tools and content with which the first version of the KnowledgeStore was populated is also provided.

Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.1	25 November 2013	Draft of Deliverable Skeleton	Francesco Corcoglioniti, Marco Rospocher	
0.2	26-27 November 2013	Draft of Introduction	Marco Rospocher	1
0.3	28-29 November 2013	Draft of Interfaces	Marco Rospocher	3
0.4	02 December 2013	Draft of Data Model	Francesco Corcoglioniti	2
0.5	09 December 2013	Draft of Hbase and Hadoop in Architecture	Roldano Cattoni	4.1.1
0.6	10 December 2013	Draft of Architecture	Francesco Corcoglioniti	4
0.7	11 December 2013	Revision of Data Model and Architecture	Francesco Corcoglioniti	2, 4
0.8	12 December 2013	Draft of RDF Populator	Francesco Corcoglioniti	5.2
0.9	13 December 2013	Draft of Executive Summary	Marco Rospocher	
1.0	13 December 2013	Draft of Background Knowledge	Francesco Corcoglioniti	5.3
1.1	16 December 2013	Revision of Hbase and Hadoop in Architecture	Roldano Cattoni	4.1.1
1.2	17 December 2013	Revision of whole Architecture	Francesco Corcoglioniti	4
1.3	17 December 2013	Added Conclusions	Marco Rospocher	6
1.4	18 December 2013	Revision of Front-End in Architecture	Francesco Corcoglioniti	4.1.3
1.5	18 December 2013	Revision of whole document	Francesco Corcoglioniti, Marco Rospocher	

Executive Summary

This deliverable documents the first implementation cycle of the NewsReader KnowledgeStore, an infrastructure for storing and reasoning about the events extracted from news, developed within the European FP7-ICT-316404 "Building structured event indexes of large volumes of financial and economic data for decision making (NewsReader)" project. The contributions presented are the results of the activities performed in Task T6.1 (KnowledgeStore internal structure) and Task 6.2 (KnowledgeStore implementation and filling) of Work Package WP6 (KnowledgeStore).

First, we introduce the idea behind the KnowledgeStore, motivating the organization of its content and presenting some examples of applications that can exploit such framework. We also highlight the key role of the KnowledgeStore in achieving the challenging goals of the NewsReader project.

We detail the KnowledgeStore, starting with a description of how unstructured (e.g., news documents) and structured (e.g., Semantic Web resources) are stored, together and in an integrated manner, within the same repository (the KnowledgeStore *data model*). We then discuss how external modules may interact with the KnowledgeStore (the KnowledgeStore *interfaces*), presenting the abstract definition and rationale of the operations through which these modules can access and manipulate the content stored in the KnowledgeStore. We also detail the internal component organization of the KnowledgeStore (the KnowledgeStore *architecture*), discussing the technological and implementation choices we made. Then, we present the KnowledgeStore *populators*, that is those tools that process annotated news documents and structured resources to fill the KnowledgeStore with content: in particular, in this first version, we filled the KnowledgeStore with selected structured resources coming from DBpedia.org, one of core repository of the Linked Data cloud.

Part of the contributions here described was presented at the 7th IEEE International Conference on Semantic Computing (ICSC2013) [Corcoglioniti *et al.*, 2013].

Contents

Table of Revisions **3**

1 Introduction **7**
1.1 The KnowledgeStore Vision . . . 7
1.2 Role of the KnowledgeStore in NewsReader . . . 10

2 The KnowledgeStore Data Model **14**
2.1 Data model design . . . 14
2.2 Data model configuration for NewsReader . . . 18

3 The KnowledgeStore Interfaces **22**
3.1 API Design Criteria . . . 22
3.2 API Operations and Endpoints . . . 24
3.2.1 CRUD Endpoint . . . 25
3.2.2 SPARQL Endpoint . . . 27

4 The KnowledgeStore Architecture and Implementation **30**
4.1 Architecture . . . 30
4.1.1 HBase & Hadoop . . . 32
4.1.2 Virtuoso . . . 33
4.1.3 Frontend Server . . . 36
4.2 Implementation . . . 36
4.2.1 Software development . . . 37
4.2.2 Deployment environments . . . 40

5 The KnowledgeStore Population **41**
5.1 NAF populator . . . 41
5.2 RDF populator . . . 43
5.3 Acquisition of LOD background knowledge . . . 45
5.3.1 Data selection . . . 45
5.3.2 Data processing . . . 49
5.3.3 Result statistics . . . 51

6 Conclusions and Future Work **55**

List of Figures

1 KnowledgeStore Content. 9
2 The role of the KnowledgeStore in NewsReader. 12
3 KnowledgeStore data model. 15
4 From RDF statements to axioms. 16
5 Representation of axioms with context and metadata using named graphs. 17
6 Example of axiom representation using named graphs. 17
7 NewsReader data model. 19
8 Invocation of CRUD retrieve operation through the HTTP ReST endpoint. 27
9 Using the KnowledgeStore client within a Java application. 28
10 SPARQL endpoint example. 28
11 KnowledgeStore architecture. 31
12 Axiom representation in HBase and in the Virtuoso Triple Store. 34
13 Examples of inference rules 35
14 Examples of generated reports on the KnowledgeStore web site. 37
15 Modular code organization. 39
16 NAF population. 42
17 Example of SPARQL query with (a) and without (b) smushing and inference. 49
18 Examples of browsing the statistics ontology in Protégé. 54

1 Introduction

This prototype deliverable presents the implementation of the first version of the KnowledgeStore [Corcoglioniti *et al.*, 2013], the infrastructure used in NewsReader to store, retrieve, and reason about the knowledge extracted from financial and economical news.

First, we present the revised version of the KnowledgeStore design, initially described in Deliverable D6.1: KnowledgeStore Design. This revision updates the KnowledgeStore design in light of the latest outcomes of some activities tightly related to the KnowledgeStore, and in particular the definition of the annotation format (D3.1: Annotation module), the NLP pipeline (D4.2.1: Event Detection – version 1), the definition of NAF[1] and the design of the whole system architecture (D2.1: System Design - draft). To favour the readability, we comprehensively describe the up-to-date version of the KnowledgeStore Data Model (Section 2), Interfaces (Section 3), and Architecture (Section 4), highlighting the main changes performed since D6.1.

We document the actual implementation of the first version of the KnowledgeStore (Section 4.2), and introduce the KnowledgeStore *populators* (Section 5), the tools supporting the filling of the KnowledgeStore with documents annotated according to NAF, and structured resources available in RDF format.

Some further content, to be considered as integral part of this deliverable, is also available as on-line resource. In particular,

- the KnowledgeStore site, which includes code, documentation (e.g., JavaDoc of the KnowledgeStore APIs), additional resources (e.g., selected DBpedia dataset), available at `http://newsreader.fbk.eu/knowledgestore`;
- the KnowledgeStore Core Data Model and NewsReader Data Model ontologies, available at `http://dkm.fbk.eu/ontologies/knowledgestore` and `http://dkm.fbk.eu/ontologies/newsreader` respectively.

The current deliverable will serve as basis for the documentation of all the next development cycles of the KnowledgeStore, and will be update and integrated to describe the reasoning service built on top of it (M24, Deliverable D6.2.2) and the final scalable version (M33, Deliverable D6.2.3).

Before going into the technical details of the KnowledgeStore, let us recall the main principles driving its development, and the let us contextualize its role within the NewsReader project.

1.1 The KnowledgeStore Vision

The rate of growth of digital data and information is nowadays continuously increasing. While the recent advances in Semantic Web Technologies (e.g., the Linked Data[2] initiative),

[1] Newsreader Annotation Format
[2] `http://linkeddata.org`

have favoured the release of large amount of data and information in structured machine-processable form (e.g., RDF dataset repositories), a huge amount of content is still available and distributed through websites, company internal Content Management System (CMS) and repositories, in an unstructured form, for instance as textual document, web pages, and multimedia material (e.g., photos, diagrams, videos). Indeed, as observed in [Gantz and Reinsel, 2011], unstructured data accounts for more than 90% of the digital universe.

Although bearing a clear dichotomy for what concern their form, the content of structured and unstructured resources is far from being separated: they both speak about *entities* of the world (e.g., persons, organizations, locations, events), their properties, and relations among them. Indeed, coinciding, contradictory, and complementary facts about these entities could be available in structured form, unstructured form, or both. Therefore, partially focusing on the content distributed in only one of these two forms may not be appropriate, as complete knowledge is a requirement for many applications, especially in situations where users have to make (potentially critical) decisions. Moreover, some applications inherently require considering both types of content: an example is *question answering* [Ferrucci *et al.*, 2010], where often a user query can only be answered by combining information in structured and unstructured sources.

Despite the last decades achievements in natural language and multimedia processing, now supporting large scale extraction of knowledge about entities of the world from unstructured digital material, frameworks enabling the seamless integration and linking of knowledge coming both from structured and unstructured content are still lacking.

This document describes the implementation of the first version of the KnowledgeStore, a framework that contributes to bridge the unstructured and structured worlds, enabling to jointly store, manage, retrieve, and semantically query, both typologies of contents. Figure 1 shows schematically how the KnowledgeStore manages these contents in its three *representation layers*. On the one hand (and similarly to a file system) the *resource layer* stores unstructured content in the form of resources (e.g., news articles, multimedia files), each having a textual or binary representation and some descriptive metadata. Information stored in this level is typically noisy, ambiguous, and redundant, with the same piece of information potentially represented in different ways in multiple resources. On the other hand, the *entity layer* is the home of structured content, that, based on Knowledge Representation and Semantic Web best practices, consists of *axioms* (a set of ⟨subject, predicate, object⟩ triples), which describe the *entities* of the world (e.g., persons, locations, events), and for which additional metadata is kept to track their provenance and to denote the formal *contexts* where they hold (e.g., in terms of time, space, point of view). Differently from the resource layer, the entity layer aims at providing a formal and concise representation of the world, abstracting from the many ways it can be encoded in natural language or in multimedia, and thus allowing the use of automated reasoning to derive new statements from asserted ones [De Bruijn and Heymans, 2007]. Between the aforementioned two layers is the *mention layer*. It indexes *mentions*, i.e., snippets of resources (e.g., some characters in a text document, some pixels in an image) that denote something of interest, such as a an entity or a axiom of the entity layer. Mentions can be automatically extracted by natural language and multimedia processing tools, that can enrich them with additional attributes

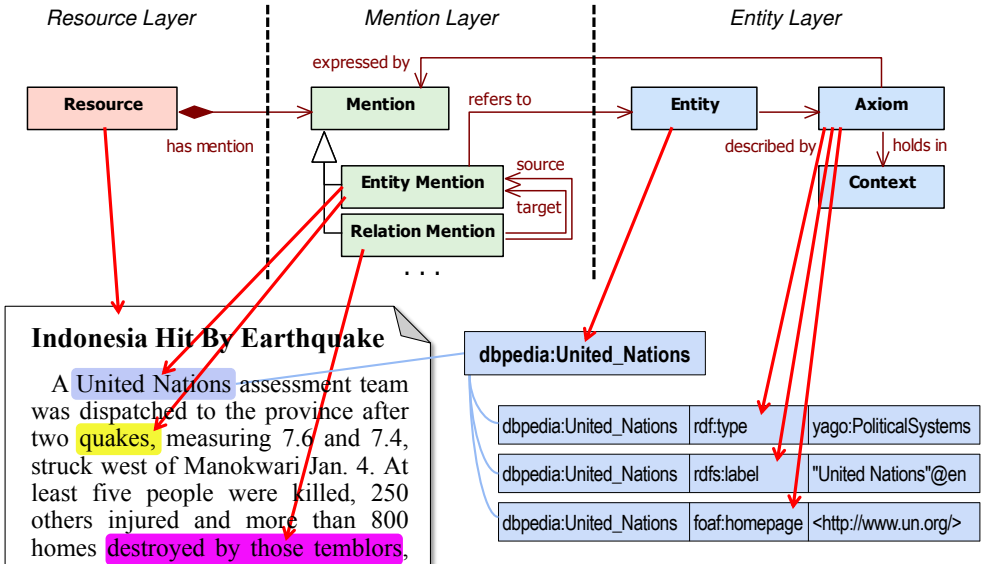

Figure 1: KnowledgeStore Content.

about how they denote their referent (e.g., with which name, qualifiers, "sentiment"). Far from being simple pointers, mentions present both unstructured and structured facets (respectively snippet and attributes) not available in the resource and entity layers alone, and are thus a valuable source of information on their own.

Thanks to the explicit representation and alignment of information at different levels, from unstructured to structured knowledge, the KnowledgeStore enables the development of enhanced applications, and favour the design and empirical investigation of several information processing tasks otherwise difficult to experiment with. To name a few:

- *Decision support.* Effective decision making support could be provided by exploiting the possibility to semantically query the content of the KnowledgeStore with requests that combine structured and unstructured content (a.k.a. mixed queries), like e.g., *retrieve all the documents mentioning that person Barack Obama participated to a sport event*—fulfilling this request involves: (i) to reason in the structured part about which events "Barack Obama" participated that are of type "sport event", and identify the corresponding participation statements; (ii) to exploit the links to the mentions those statements have been extracted from; and (iii) to exploit the linking between those mentions and the resources containing them [Hoffart *et al.*, 2011].

- *Coreference resolution.* The KnowledgeStore favours the implementation and evaluation of tools which exploit available structured knowledge to improve the performance of coreference resolution tasks (i.e., identifying that two mentions refer to the same entity of the world), as shown in [Bryl *et al.*, 2010], especially in cross-document / cross-resource settings.

- *Ontology population.* Finally, the joint storage of extracted knowledge, the resources

it derives from, and extraction metadata provides an ideal scenario for developing, training, and evaluating ontology population [Buitelaar and Cimiano, 2008] techniques. In particular, the KnowledgeStore data model favours the exploration of a number of computational strategies for *knowledge fusion*, i.e., the merging of possibly contradicting information extracted from different sources, and *knowledge crystallization*, i.e., the process through which information from a stream of multimedia documents is automatically extracted, compared, and finally integrated into background knowledge, taking into consideration how many times a piece of information has been extracted, where it has been extracted from and how well it fits / is consistent with pre-existing background knowledge.

Given the KnowledgeStore ambition to cope with a huge quantity of data and resources (potentially in the range of billions of documents), as required by today / next future applications, the development of the KnowledgeStore vision is necessarily driven by *scalability* aspects: performances in storing, accessing, and querying the KnowledgeStore have to gracefully scale with respect to the size of managed content. For this reason the implementation of the KnowledgeStore is based on technologies compliant with the deployment in distributed hardware settings, like clusters and cloud computing.

The idea behind the KnowledgeStore was preliminary investigated in [Cattoni *et al.*, 2012] and tested in the scope of the LiveMemories project[3]. However, we highly revised the design of the previous version, introducing significant enhancements: this first new version of the KnowledgeStore supports (i) the storing of and reasoning on events and related information, such as event relations (the previous version was limited to mentions and entities referring to persons, organizations, geo-political entities, and locations), (ii) scaling on a significantly larger collection of resources, and (iii) a semantic query mechanism over its content, to favour the development of reasoning services on top of it (no reasoning services was previously offered).

1.2 Role of the KnowledgeStore in NewsReader

The goal of the NewsReader Project[4] is to process daily economical and financial news in order to extract events (i.e., what happened to whom, when and where – e.g., "The Black Tuesday, on 24th of October 1929, when United States stock market lost 11% of its value"), and to organize these events in coherent narrative stories, combining new events with past events and background information. These stories are then offered to professional decision-makers, that by means of visual interfaces and interaction mechanisms will be able to explore them, exploiting their explanatory power and their systematic structural implications, to make well-informed decisions. Achieving these challenging goals requires:

- to process document resources, detecting mentions of events, event participants (e.g., persons, organizations), locations, time expressions, and so on;

[3] `http://www.livememories.org/`
[4] `http://www.newsreader-project.eu/`

- to link extracted mentions with entities, either previously extracted or available in some structured domain source, and coreferring mentions of the same entity;
- to complete entity descriptions by complementing extracted mention information with available structured knowledge (e.g., DBPedia[5], corporate databases);
- to interrelate entities (events and their participants, in particular) to support the construction of narrative stories;
- to reason over events to check consistency, completeness, factuality and relevance;
- to store all this huge quantity of information (on resources, mentions, entities) in a scalable way, enabling efficient retrieval and intelligent queries;
- to effectively offer narrative stories to decision makers.

A framework like the KnowledgeStore can effectively contribute to address such kind of requirements[6].

First, the KnowledgeStore allows to store in its three interconnected layers all the typologies of content that have to be processed and produced when dealing with unstructured content and structured knowledge:

- the *resource layer* stores the unstructured financial news and their annotations;
- the *mention layer* identifies fragments of news denoting entities (e.g., a take-over event), relation between entity mentions (e.g., event participation), numerical quantities (e.g., a share price);
- the *entity layer*[7] stores the structured descriptions of those entities extracted from resources and merged with available structured knowledge (e.g., Linked Data sources, corporate databases).

Second, as shown in Figure 2, the KnowledgeStore acts as a shared data space supporting the interaction of the several NewsReader modules and tools envisaged according to the aforementioned requirements: modules retrieve their input data from the KnowledgeStore, and store the results of their processing back in it, so that they can be picked up by other modules. Modules can be roughly classified in five categories:

- *News* and *RDF populators*. These modules, developed as part of WP6 activities, enable the bulk loading of structured and unstructured contents in the KnowledgeStore. The former processes a collection of NAF annotated news documents injecting content in all three layers of the KnowledgeStore, while the latter augments the entity layer with Semantic Web compliant resources available in RDF repositories.

[5] `http://dbpedia.org/`

[6] Note that such requirements, though arisen from the specific application scenario considered within the NewsReader project, are quite typical in many application contexts where enhanced applications (e.g., decision support systems, information retrieval systems, semantic search engines, query answering applications) have to deal with both unstructured content and structured knowledge.

[7] In the current status of affairs, an ad-hoc layer to explicitly represent narrative stories is not foreseen. Narrative stories will be represented within the entity layer, by means of entities and statements.

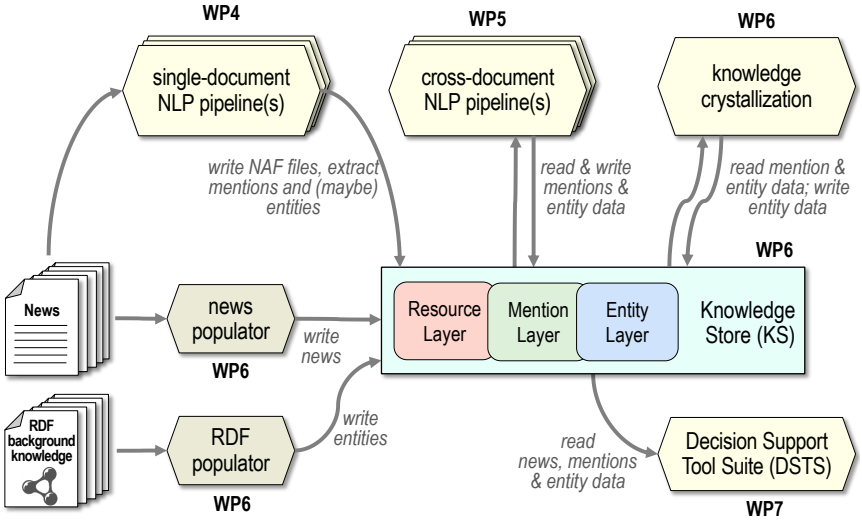

Figure 2: The role of the KnowledgeStore in NewsReader.

- *single-document NLP pipelines.* These pipelines, as part of WP4 activities, work at the resource layer, and take care of processing a text document enriching it with linguistic annotations related to tokenization, Part-Of-Speech (POS) tagging, Word Sense Disambiguation (WSD), named entity and event recognition, semantic role labelling, and so on.

- *cross-document NLP pipelines.* These modules, as part of WP5 activities, work at the mention and entity layers, exploiting the work of the NLP pipelines to instantiate, link, or enrich entities performing tasks such as cross-document coreference.

- *knowledge crystallization tools.* These modules, as part of WP6 activities, will compare and merge the information extracted by the WP4 and WP5 pipelines, finally integrating it into the background consolidated knowledge.

- *Decision Support Tool Suite (DSTS).* Finally, as part of WP7 activities, the decision support tool suite queries the KnowledgeStore— mainly the entity layer (although queries may also requires to retrieve documents and mentions)—to obtain the information about events and narrative stories to be shown to users.

The KnowledgeStore provides to external modules different typologies of access to its content: *create, read, update, delete* (CRUD) operations on resource/mention/entity/statement, and retrieve/query mechanisms. Due to the goals of the NewsReader project, the development of the KnowledgeStore implementation focuses on providing efficient retrieve/query mechanisms; still, a basic implementation of all the CRUD operations is provided, such that external modules have full access (and control) on the content of the KnowledgeStore[8].

[8]Note that some operations on a single element of the KnowledgeStore content may also impact on other elements (e.g., deletion of a news in the resource layer affects the mentions associated to that news, which

The NewsReader technologies will be assessed with economic and financial news and on events relevant for political and financial decision-makers. Concerning the data and information volume aspect, this is a quite significant domain. Roughly 25% of the news deals with finance and economy, and a large international information broker such as the project partner LexisNexis, typically handles about 2 million news each day, cumulating to an impressive 25 billion documents archive spanning several decades. As suggested by these numbers, the project context sets an ideal test bed to assess the scalability of the KnowledgeStore.

may affect entities associated to those mentions). The correct handling of these situations is not clear, and has to be investigated. Therefore the KnowledgeStore does not handle them, although it offers to each module the basic operations to implement the more appropriate strategy to cope with them.

2 The KnowledgeStore Data Model

Changes wrt the KnowledgeStore Data Model described Deliverable D6.1

- entities described by *axioms* (instead of plain RDF statements) to support appropriate TBox storing; each *axiom* models a contextualized, annotated logical axiom encoded by one or more RDF statements (Section 2.1);
- alignment of data model concepts to Dolce+DNS Ultralite ontology and partial renaming for consistency with Dolce terminology (Section 2.1);
- adaptation of data model to revised WP3 annotation guidelines latest WP2 specification of the NewsReader Annotation Format (NAF) (Section 2.2).

The data model defines what information can be stored in the KnowledgeStore, in accordance with the annotation guidelines of WP3[9], the event modelling activity of WP5 and the NewsReader Annotation Format (NAF) of WP2[10]. It serves both as a basis for the design of the KnowledgeStore, and as a shared model that permits WP4 and WP5 linguistic processors and the decision support tool suite of WP7 to cooperate.

Flexibility is a key requirement of the data model, given its role. This is addressed through the design of a minimalist, configurable data model, centred around the key concepts of resource, mention and entity described by axioms within a context. The data model is then configured for use in NewsReader (but also other scenarios) through the controlled addition of attributes, relations, and resource and mention sub-types.

The remainder of this section provides an high-level description of the KnowledgeStore data model (Section 2.1) and its configuration for NewsReader (Section 2.2), while their specifications are available online on the KnowledgeStore documentation site. The presentation is at a conceptual level with no implication on the physical organization of data.

2.1 Data model design

The KnowledgeStore data model is depicted in the UML class diagram of Figure 3. The model is organized in the three *resource*, *mention* and *entity* layers and consists of a *fixed* part and a *configurable* one, as highlighted in the figure. Both parts are specified as OWL 2 ontologies [Motik *et al.*, 2009] (available online) containing the TBox definitions and restrictions for each model element. The first ontology for the fixed part is embodied in the KnowledgeStore implementation, while the the latter is supplied at configuration time and exploited to fine tune the system and, in perspective, to enable additional services such as data validation that might be added in next releases of the KnowledgeStore. The grounding of the data model in OWL 2 ontologies allow to encode both the model and its

[9]The revised guidelines will be described in Deliverable D3.3.1: Annotated Data.

[10]NAF (Newsreader Annotation Format) is the format adopted in the project to augment resources with structured information extracted by linguistic processors (tokenization, POS tagging, Semantic Role labelling, and much more). NAF will be described in Deliverable D2.1: System Design.

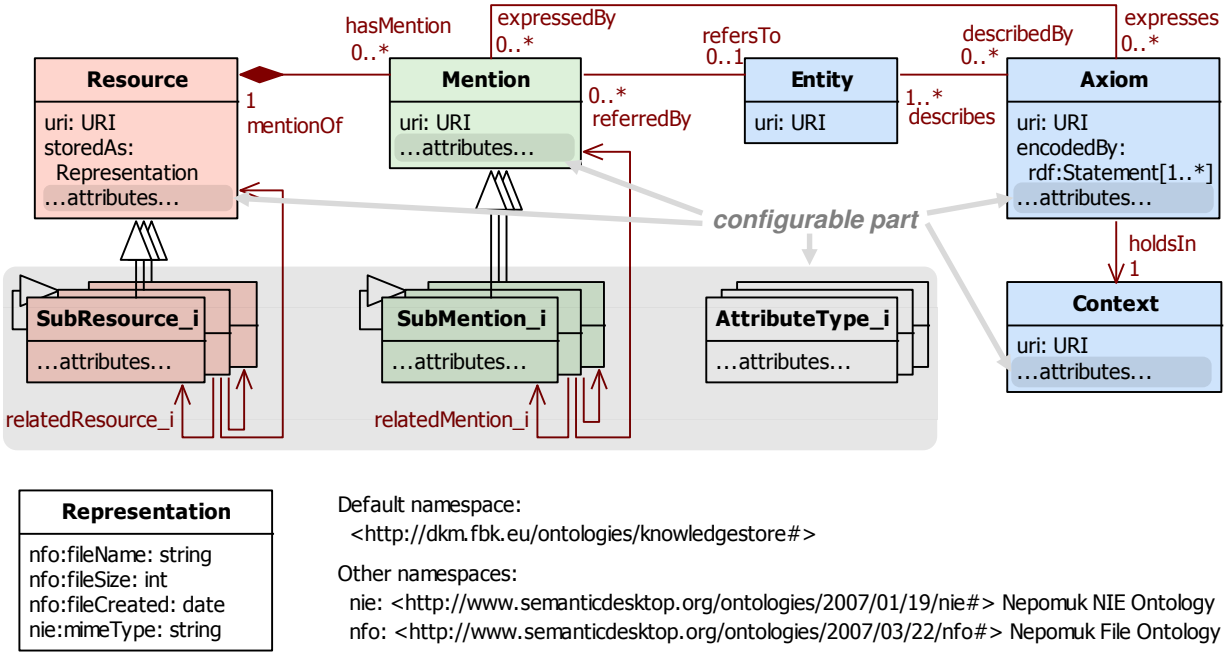

Figure 3: KnowledgeStore data model.

instance data in RDF [Beckett, 2004], which in turn enables interoperability with Semantic Web applications and technologies.

Fixed part This part defines the core abstraction of the model. It is formalized in a KnowledgeStore OWL 2 ontology[11] by reusing terms from external vocabularies and providing alignments to concepts in the Dolce+DNS Ultralite upper ontology[12]. It includes:

- The Resource, Mention, Entity core classes. Their instances are described using the types, attributes and relations defined in the configurable part of the model; they are identified by externally-assigned uris, set at creation time and then immutable.
- The core relations among these three classes: a Resource has Mentions, and each Mention may refersTo an Entity.
- The files storing resource representations and their metadata managed by the system (storedAs attribute and Representation class).
- The Axiom and Context abstraction used to provide open descriptions of entities. An Axiom is a logical formula (e.g., that "Barack Obama is president of USA") that is encoded with one or more RDF statements and that possibly holdsIn a specific Context (e.g., the time period 2009-2016). Both axioms and contexts are identified with URIs automatically assigned by the system based on the RDF statements and

[11] http://dkm.fbk.eu/ontologies/knowledgestore
[12] http://ontologydesignpatterns.org/wiki/Ontology:DOLCE%2BDnS_Ultralite

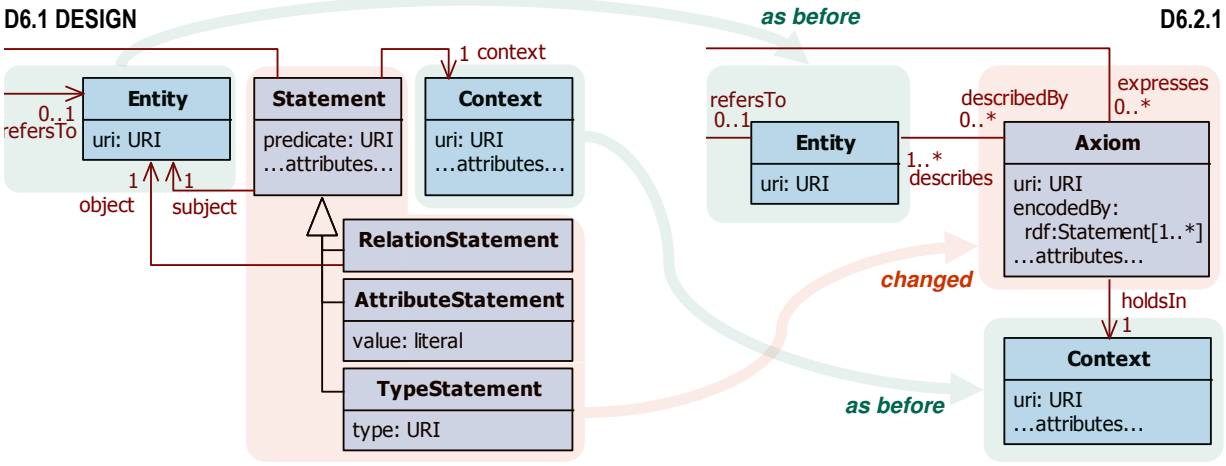

Figure 4: From RDF statements to axioms.

context of the former and the contextual attributes of the latter (which are defined in the configurable part).

- The relations describes and expressedBy linking an axiom, respectively, to the entities it describes and the mention it has been extracted from, if any; the latter information is relevant both for external users (e.g., decision makers) and for debugging an information extraction pipeline built on top of the KnowledgeStore.

Being embodied in the implementation, the fixed part of the model is kept as small as possible in order not to sacrifice flexibility. Therefore, relevant information such as resource metadata, contextual dimensions, mention types and linguistic attributes are not defined in this part, due to the fact that a stable, exhaustive and cross-domain characterization of them cannot be drawn; this information can however be added to the configurable part and tuned to the representation needs of a particular scenario (such as NewsReader).

The representation of axioms in place of plain RDF statements represents the major change from the data model described in Deliverable D6.1 (other changes are the alignment of some concepts to Dolce and their renaming to make them more consistent with Dolce terminology). The design of D6.1 directly associated context and metadata to RDF statements, under the assumption that each RDF statement was a logical axiom. While this assumption holds for ABox assertions, we realized that data in the KnowledgeStore may also comprise complex TBox axioms whose encoding requires multiple RDF statements (e.g., an OWL class restriction). Associating context and metadata to each of those statement is conceptually wrong, inefficient and a potential source of problems (in case different statements of the same axiom are associated to different context or metadata). This motivated a revision of the model, adding Axioms as first class citizens; the change from D6.1 statements to D6.2.1 axioms is specifically illustrated in Figure 4.

While axioms are just bunches of triples that can be encoded with plain RDF, axiom metadata and contextual information are more complex to represent in RDF; still, their

```
@prefix ckr: <http://dkm.fbk.eu/ckr/meta#> .
<module_uri> { ... axiom triples ... }
ckr:global {
  <module_uri>  <metadata_property_1> <metadata_value_1> ; ... ;
                <metadata_property_N> <metadata_value_N> .
  <context_uri> a ckr:Context ;
                ckr:hasModule <module_uri> ;
                <contextual_dimension_1> <contextual_value_1> ; ... ;
                <contextual_dimension_M> <contextual_value_M> ;
}
```

Figure 5: Representation of axioms with context and metadata using named graphs.

```
# ckr, ks, nwr, sem, dbo, ex, dbpedia, dbo, xsd prefix definitions omitted
ex:mod01 { dbpedia:Barack_Obama dbo:birthPlace dbpedia:Honolulu }  # the axiom
ckr:global {
  ex:mod01 nwr:crystallized "true"^^xsd:boolean ;
           nwr:confidence 1.0 ;
           nwr:source dbpedia:DBPedia ;                    # comes from DBPedia
           ks:expressedBy ex:mention15 , ex:mention127 . # but also extracted from
                                                          # two mentions
  ex:ctx01 a ckr:Context ;
           ckr:hasModule ex:mod01 ;
           sem:hasTimeValidity ex:any-time ;  # open interval, definition omitted
           sem:hasPointOfView ex:common-pov . # express common POV without particular
                                              # authority, definition omitted
}
```

Figure 6: Example of axiom representation using named graphs.

RDF representation is a requirement for enabling import and export of RDF entity data and thus making the KnowledgeStore compatible with existing RDF datasets. We address this issue using *named graphs* [Carroll *et al.*, 2005], an extension of RDF supported by the majority of tools and by several RDF syntaxes, and following and extending the CKR approach [Bozzato and Serafini, 2013]. Using named graphs, an axiom together with its context and metadata can be represented as shown in Figure 5: the triples encoding the axiom are stored in a graph called *module*, which in turn is associated to the axiom metadata inside special ckr:global graph; contextual information is also encoded in ckr:global, and attached to the axiom module via a ckr:hasModule triple. A concrete example of this representation is shown in Figure 6. While seemingly verbose, this representation allows putting multiple axioms in the same module in case they share the same context and metadata (this is often the case for axioms coming from the same source), thus limiting the number of triples in ckr:global and making the associated overhead negligible.

Configurable part This part is specified at configuration time and is available both to the KnowledgeStore and to its users, acting as the reference schema against which queries and other data access operations can be formulated. It includes:

- The subclass hierarchy of Resource and Mention (entities excluded as described via axioms); subclasses are not assumed to be disjoint.

- The additional attributes of Resource, Mention, Axiom, Context and their subclasses. Context attributes define the contextual dimensions for a particular scenario and are used by the system to generate the context URI. In case of objects belonging to multiple subclasses, their description can make use of all their combined attributes.

- Additional relations among resources or among mentions (but not between the two).

- Enumerations and classes used as attribute types (similarly to ks:Representation).

- Restrictions on the domain and range of fixed-part relations (not shown in figure).

2.2 Data model configuration for NewsReader

The UML class diagram in Figure 7 shows the latest[13] configuration of the data model for NewsReader. With respect to the configuration described in Deliverable D6.1, the version here described has been revised to take into consideration the revised annotation guidelines of WP3 (to be described in Deliverable D3.3.1: Annotated Data) as well as the latest NAF specification (to be described in Deliverable D2.1: System Design). The OWL 2 ontology formally encoding the model is available online[14]. In the following, an overview of the resulting model is presented, proceeding along the three *resource*, *mention* and *entity* layers (note that URIs are hereafter abbreviated using qualified names and a default NewsReader data model namespace).

Resource layer For each processed news, two resources are stored in the KnowledgeStore: (i) a News resource for the news itself, containing its metadata and, optionally, its textual content (depending on availability and copyright agreements); and (ii) a NAFDocument resource storing the NAF document generated for the news. More in details:

- News are described using metadata from the Dublin Core Metadata Terms vocabulary (dct:* attributes), augmented with NewsReader-specific attributes to keep track of the external source document the news has been imported from (originalFileName, originalFileFormat, originalPages, as defined in NAF).

- NAF documents are described with the subset of metadata from the NAF header that is most relevant for selecting NAF documents in the KnowledgeStore. This subset comprises the NAF version, the publicId of the NAF document (attribute dct:identifier), the NAF layers available in the NAF document (e.g., text, terms, deps), the NAF processors used (dct:creator) and the language of the processed document (dct:language); complete metadata and all the produced linguistic annotations are available in the stored XML content of the NAF document.

[13] As of 2013/12/15. Minor changes may occur to best accommodate the NAF output of WP4 pipeline.
[14] `http://dkm.fbk.eu/ontologies/newsreader`

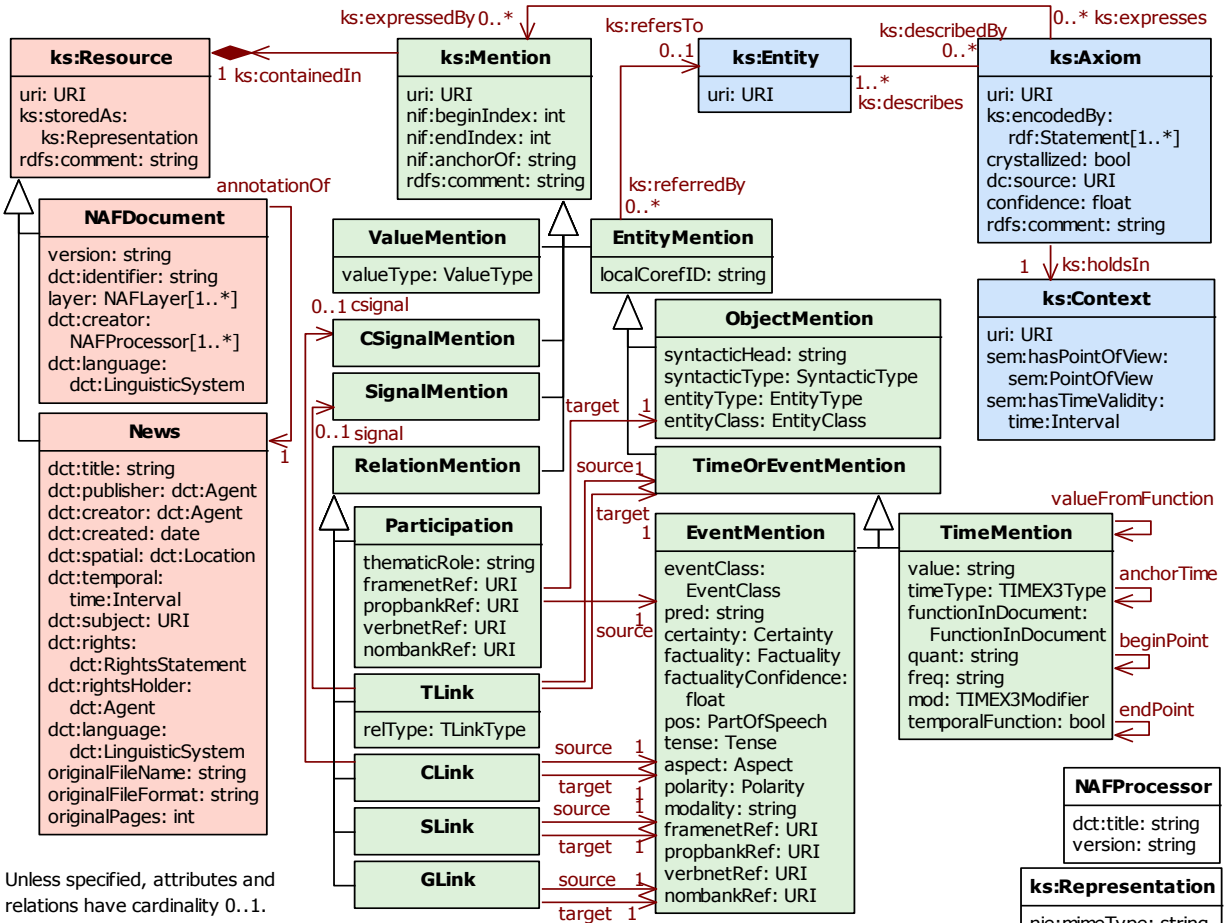

Figure 7: NewsReader data model.

Mention layer The position of a mention in a news is encoded with numerical character offsets based on the NLP Interchange Format (NIF) vocabulary[15] (nif:beginIndex, nif:endIndex, nif:anchorOf), so to enable interoperability with tools consuming NIF data. Four main types of mentions are distinguished:

- *Entity mentions* denote entities in the domain of discourse (linked with refersTo). An optional localCorefID attribute can be used to group mentions coreferring within a document (intra-document coreference). Entity mentions are further characterized based on the type of entity:
  - Object mentions refer to persons, locations, organizations, products, financial objects (e.g., "NASDAQ Index") and mixed entities (e.g., "the CEO and his company"), discriminated via attribute entityType; the types considered are those proposed in the revised annotation guidelines of WP3. Object mentions are described by a syntactic head, a syntactic type (e.g., name, nominal or pronoun) and a linguistic entity class (e.g., specific referential).
  - Time mentions are described using the subset of TIMEX3 properties selected in NAF and in the annotation guidelines. These properties include: the TIMEX3 type (e.g., date, time, duration); the normalized time value; the function within the document (e.g., document creation time); relations with other time mentions (beginPoint, endPoint, anchorTime, valueFromFunction); the optional quantifier (e.g., "every"), frequency (e.g., twice-a-month) and modifier (e.g., "approx") characterizing the expression and whether it is used as a temporal function.
  - Event mentions are characterized using a number of attributes: the linguistic class of the event (e.g., speech-cognitive); the lemma of the token conveying the event (pred); the part-of-speech (pos), e.g., adjective, noun or verb; the certainty and factuality of the event, together with a factuality confidence; the tense, aspect, polarity (e.g., positive) and modality (e.g., "should") of the verbal form used. In addition, references to external resources further specifying the type of event are stored (framenetRef, verbnetRef, propbankRef,nombankRef).
- *Relation mentions* express relations between two entities, whose mentions are identified by source and target links. Different kinds of relation mentions are stored:
  - Causal links (CLink) express a causal relation between two events.
  - Temporal links (TLink) denote a certain temporal relation (relType, e.g., before, include, simultaneous) among two events or time expressions.
  - Subordinate links (SLink) express certain structural relations among events.
  - GLinks express grammatical relations among events (as in "the share drop came on the same day", with "drop" and "came" being events).

[15] `http://nlp2rdf.org/nif-1-0`

  - Participation mentions denote the participation of an entity to an event in a certain thematic role (semRole), possibly further specified by references to external resources (framenetRef, verbnetRef, propbankRef,nombankRef).

- *Signal and CSignal mentions* identify pieces of text supporting the existence of a temporal or causal relation, to which they are linked by relations signal.

- *Value mentions* are numerical expressions used for quantities (cardinal numbers in general), percentages and monetary expressions; the type of value is stored.

Entity layer Different kinds of entities are stored, including persons, organizations, geopolitical entities or locations, events, points and intervals in time extracted from text; the type of entity is conveyed by an rdf:type axiom. The context in which an axiom holds is described and identified in terms of temporal validity (sem:hasTimeValidity) and time-referenced point of view (sem:hasPointOfView, e.g., "Financial Times" point of view expressed on 2013/12/15); the Simple Event Model (SEM) [van Hage *et al.*, 2011] and the OWL Time[16] vocabularies are used to that purpose. Axiom metadata consists of a confidence value (confidence), a provenance indication (dct:source) and a crystallized flag (crystallized). Confidence is represented on a $0.0 - 1.0$ scale and quantifies how reliable an extracted statement is. Provenance is stored for background knowledge axioms and denote the external sources they have been imported from (e.g., DBPedia).[17] The crystallized flag is set for axioms belonging to background knowledge or assimilated to it after repeated extraction of the conveyed information, according to some crystallization algorithm. This algorithm (to be defined as part of WP6 T6.2) will exploit information such as how many mentions a statement has been extracted from (attribute ks:extractedFrom) and in which time frame, as well as which resources (e.g., which kind of news) it was extracted from and how reliably; it will also consider pre-existing background knowledge, in form of TBox constraints and other ABox assertions an axiom should be consistent with.

[16] `http://www.w3.org/TR/owl-time/`

[17] The adoption of a provenance model to track sources, authority, and tool processing activities, is still under definition at project level at the time of writing this deliverable. The data model here presented might thus be revised according to the resulting provenance model.

3 The KnowledgeStore Interfaces

Changes wrt the KnowledgeStore Interfaces described Deliverable D6.1

- revision of the API design criteria, to document some implementation choices adopted (Section 3.1);
- revised the organization of the API operations in two main KnowledgeStore endpoints: CRUD and SPARQL (Section 3.2); introduced some examples of their usage;
- introduced a description of the KnowledgeStore Java API client, with a code example.

The KnowledgeStore presents a number of interfaces, offered as part of the KnowledgeStore API, through which external clients may access and manipulate stored data. In this section we present their abstract definition and their rationale. In particular, Section 3.1 recalls the criteria underlying the design of the API, while Section 3.2 presents an overview of the operations offered thorough it: two main categories of operations are described, together with some representative examples. The Java API documentation describing the full list of operations offered by the KnowledgeStore is available online[18].

3.1 API Design Criteria

When designing the API of a complex system such as the KnowledgeStore, a number of aspects have to be considered carefully. Those aspects, and the solutions adopted for the implementation of the first version of the KnowledgeStore, are discussed in the following.

Operation granularity An API may offer fine-grained, elementary operations operating on single objects (e.g., a single mention update), as well as coarse-grained operation that operate on whole sets of objects at a time (e.g., the simultaneous update of all the mentions of a certain resource). Fine-grained operation may be inefficient, as modifying a set of objects requires multiple API calls with the associated overhead; on the other hand, a coarse-grained approach may result in a complex API with a larger number of (similar, overlapping) operations due to the need to provide different ways to select the objects to operate on (e.g., update all the mentions of a given type, with a certain attribute, with specific identifiers, ...). In the first release of the KnowledgeStore, we address this issue by offering efficient coarse-grained operations that operates on multiple objects at once (borderline case: a single object), but at the same time we introduce an XPath based selection language to provide clients with a flexible way to select the objects to operate on, therefore avoiding an explosion of the number of API operations.

[18] `http://newsreader.fbk.eu/knowledgestore/`

Message exchange pattern API operations may work according to a synchronous *request-response* pattern (the client issues the request and waits for its reply), or according to asynchronous *message exchange patterns* such as *asynchronous polling* (the client issues a request and polls repeatedly the server about the status of the operation) or *asynchronous notification* (the client issues the request and is later notified by the server when the processing is finished). The *request-response* pattern is simpler for clients and will be used in the implementation of the first version of the KnowledgeStore. Asynchronous approaches cope better with long running API operations, as they avoid timeout issues at the various network protocol levels: based on the experience gathered with this release of the KnowledgeStore, we will evaluate whether to investigate and possibly support also asynchronous message exchange patterns for selected API operations.

Transactional properties Transactions are units of work—either a single operation or a sequence of operations—to which certain properties are associated, such as the *ACID* properties of relational databases: *atomicity*, *consistency*, *isolation* and *durability*.[19] Unfortunately, enforcing ACID properties in distributed, scalable systems like the KnowledgeStore is difficult, inefficient and even theoretically impossible in case system *availability* (i.e., the fact every request is answered) is also desired. With this premise, and assuming the need for *partition-tolerance* (due to the distributed nature of the system), the CAP theorem [Gilbert and Lynch, 2002] rules out consistency, and thus ACID in a strict sense.[20] The situation asks for a trade-off solution, that for the KnowledgeStore may favour consistency and ACID properties over availability, on the basis that it is deemed preferable for a client request to fail (in presence of nodes or network failures) rather than returning stale data. In the first KnowledgeStore release, a coarse-grained API call will behave in a transactional way and satisfy ACID properties on each single object handled in the call (e.g., a single element in a set of mentions), as this can greatly simplify writing client applications. This means that each object in the set of objects modified by an API call will be either successfully modified or not modified at all (atomicity); if modified, the new state of the object will be valid (consistency) and permanently stored (durability), and no concurrent client will see intermediate states during the modification of the object (isolation). If feasible, further developments may support the explicit delimitation of transactions by clients through the introduction of begin and end transaction operations.

Data validation The specialized data model (see Section 2.2) defines a number of constraints that must be satisfied by data both stored in the system and received in input to API operations. In the first release of the KnowledgeStore, essential data validation on input data is performed for each API request, in order to check the preconditions which are instrumental to the successful completion of the operation (e.g., presence and validity of object identifier and mandatory attributes). However, the KnowledgeStore design is com-

[19] `http://en.wikipedia.org/wiki/ACID`

[20] *Eventual consistency*, i.e., the fact the system will eventually become consistent in absence of inputs, is permitted; still, this is a weak form of consistency that has to be taken into consideration by applications.

patible with more expressive data validation solutions, that may be implemented in future releases by exploiting the OWL 2 roots of the data model for declaring and validating complex constraints;[21] violations of these constraints may either be reported as warnings or may cause the API request to fail.

Security Access to the KnowledgeStore API must be restricted only to authorized clients, since it allows the modification of stored contents and the retrieval of possibly copyrighted or otherwise access-restricted information (e.g., news articles accessible only for research purposes). As it is conceivable for the KnowledgeStore API to be made accessible over an unprotected channel such as the Internet, the first release of the KnowledgeStore implements suitable technical measures at the API level to enforce client authentication and to selectively encrypt the exchange of sensitive data. Authentication is based on separate username/password credentials for each authorized client. Authenticated clients may read all the contents stored in the KnowledgeStore, possibly with some limitations in terms of throughput and number per day of read operations (in order to enforce a fair use of the system); selected clients are also granted write permission on all the stored contents.

3.2 API Operations and Endpoints

To define the operations to be implemented by the KnowledgeStore, all technical partners of the consortium were asked to analyse the kind of content their modules were expected to obtain/inject in it, and how. For this purpose, partners were asked to fill in a template on a page in the project CMS[22] with information on operations they were expecting to use to interact with the KnowledgeStore. For each operation, they were required to provide:

- a name;
- a description explaining the rationale of the operation;
- the input parameters used to invoke the operation;
- the expected output returned by the operation;
- some examples of usage of the operation;
- possible observations about the operation (e.g., optional attributes, or variants);

The collected operations were then first analysed[23] to find commonalities, in order to remove duplicates or operations subsumed by other ones. By adopting a generalization

[21] In this case, the *open world assumption* (OWA) underlying OWL 2 and its rejection of the *unique name assumption* (UNA) must be taken into consideration. Under OWA, missing mandatory information is inferred rather than being reported as a constraint violation. This is undesirable for data known to be complete (e.g., certain resource and mention metadata), in which case OWL 2 extensions such as the ones presented in [Patel-Schneider and Franconi, 2012] or in [Tao *et al.*, 2010] can be adopted. Concerning UNA, it holds for the objects managed by the KnowledgeStore. By ignoring it, functionality restrictions over properties of those objects will infer their equivalence, rather than detect a constraint violation. This can be fixed by automatically declaring objects in the KnowledgeStore as owl:differentFrom each other.

[22] Accessible from the KnowledgeStore website: `https://newsreader.fbk.eu/knowledgestore`

[23] The analysis here described refers to the content of the operations CMS page as of 15.12.2013; the page may evolve as additional operations are requested by the processing modules being developed.

perspective, to favour an easy deployment of the KnowledgeStore in broader application scenarios that the scope of NewsReader, we also replaced some of the collected operations with new ones subsuming them. The full list of resulting operations is described in the project CMS[24].These operations are offered to the users as part of the KnowledgeStore API through two endpoints: the *CRUD endpoint*, that provides the basic operations to access and manipulate the objects stored in all the layers the KnowledgeStore, and the *SPARQL endpoint*, that enables flexible access to the semantic content store in the entity layer. Here below, we present a brief overview of these endpoints and the operations they support.

3.2.1 CRUD Endpoint

The CRUD endpoint provides the basic operations to access and manipulate (CRUD: *create*, *retrieve*, *update*, and *delete*) any object stored in any of the layers of the KnowledgeStore. Operations of the CRUD endpoint are all defined in terms of sets of objects, in order to enable bulk operations as well as operations on single objects. In details, the following operations are provided for resources, mentions, entities and axioms:

- `create (object descriptions) : assigned URIs and/or creation errors`
  Stores new objects based on their supplied descriptions. Object URIs are supplied by the client (differently from D6.1 design). Due to data validation, creation may succeed only for a subset of objects; for the remaining objects no data is stored and the corresponding URIs and errors are reported to the client. As a large number of objects may be created in a single call, input descriptions are streamed to the server, while per-object success or error acknowledgments are streamed back to the client.

- `retrieve (condition, output attributes) : object descriptions`
  Returns all the objects matching a supplied XPath-like *condition*. The condition can select objects based on a number of criteria over object types and attributes, possibly considering complex nested properties (e.g., /ks:storedAs/nie:mimeType = 'text/plain' can be used to select all the resources having a plain text representation[25]). Results are reported in no particular order and include either all the objects' attributes or only the specified set of object attributes (if non-empty). Results are streamed to the client, that can consume them as they arrive.

- `update (condition, object description, merge criteria) : update errors`
  Updates all the objects matching a supplied condition, setting one or more of their attributes to a particular value; if the attributes were already set, *merge criteria* can be optionally used to combine old values with new ones (e.g., overwrite, take the union of the two, ...). This operation mirrors the corresponding SQL `update` command and permits to efficiently clear or set one or more attributes on an unbound set of objects, avoiding the overhead of first retrieving the objects to modify and then updating their attributes one object at a time. Similarly to `create`, it is possible

[24] Accessible from the KnowledgeStore website: `https://newsreader.fbk.eu/knowledgestore`
[25] Please refer to the online documentation for the full condition syntax.

that only a subset of the objects is updated (e.g., because of data validation); for the remaining objects, URIs and errors are reported to the client.

- `delete (condition) : deletion errors`
  Deletes all the objects matching a supplied condition. Note that objects on which other objects depend (e.g., a resource referenced by some mention) cannot be deleted. Therefore, it is possible for the operation to delete only a subset of the matching objects; for the remaining objects, URIs and errors are reported to the client.

- `merge (object descriptions, merge criteria) : merge errors`
  Updates a set of objects given their identifiers, setting one or more attributes (or entity axioms) to specific values and possibly applying merge criteria to combine old and new values. The operation is idempotent and provides an additional way to update existing data, supporting the common use case where a bunch of objects is processed (e.g., by an NLP module) resulting in new attributes being computed, and the resulting local descriptions have to be merged back with the complete descriptions in the KnowledgeStore. Note that merging may succeed only for a subset of objects (because of data validation or change of unmodifiable attributes); for non-merged objects, URIs and errors are reported to the client.

- `count (condition) : # matching objects`
  Returns the number of objects matching a supplied condition. The operation is strictly redundant as it can be implemented based on `retrieve`; nevertheless, it is defined in order to avoid the retrieval of huge quantities of data from the KnowledgeStore when just a count is needed. This operation might be replaced by a more general `aggregate()` operation in future versions of the KnowledgeStore.

While all the above operations work on objects of the same kind (on a single call), the CRUD endpoint offers also retrieval operations that affects objects from different layers of the KnowledgeStore. An example, is the general-purpose `match` operation:

- `match (condition and output attribute URIs at resource, mention, entity and axiom levels) : matching <resource, mention, entity, axiom> 4-tuples`
  Returns a set of ⟨resource, mention, entity, axioms⟩ 4-tuples whose mention occurs in the resource, refers to the entity and supports the extraction of the axioms, and such that the attributes on all the four components satisfy the specified conditions; for each tuple, a specified set of output attributes for the four components is returned.[26]

The CRUD endpoint is made available to external KnowledgeStore users in two modalities: through an HTTP ReST Server, and as a Java client: the former favours the integration of the KnowledgeStore in complex frameworks where tools developed with different technologies are deployed; the latter, actually built on top of the former, enables the easy

[26] With respect to D6.1 design, the *axioms* component has been added to address a new requirement from the decision support tool suite of WP7.

```
curl -request GET http://newsreader.fbk.eu/kstest/resources.rdf?$where=
   dct:publisher = dbpedia:TechCrunch (*)
```

(*) URL encoding omitted

```
<rdf:RDF xmlns:nwr="http://dkm.fbk.eu/ontologies/newsreader#" ...>
  <nwr:News rdf:about="http://newsreader.fbk.eu/resources.rdf/r105">
    <dcterms:title>Salesforce Is A Platform Company. Period.</dcterms:title>
    <dcterms:publisher rdf:resource="http://dbpedia.org/resource/TechCrunch" />
    <dcterms:issued>2013-09-30</dcterms:issued>
    <nfo:fileURL>http://techcrunch.com/2013/09/30/...</nfo:fileURL>
    <nie:isStoredAs rdf:resource="http://newsreader.fbk.eu/resources.rdf/r105.txt">
      <nfo:fileName>r105.txt</nfo:fileName>
      <nfo:fileSize>15012</nfo:fileSize>
      <nfo:fileCreated>2013-09-30</nfo:fileCreated>
      <nie:mimeType>text/plain</nie:mimeType>
    </nie:isStoredAs>
  </nwr:News>
  ...
</rdf:RDF>
```

Figure 8: Invocation of CRUD retrieve operation through the HTTP ReST endpoint.

integration in Java-based tools. Figure 8 shows the invocation through the HTTP ReST CRUD endpoint of a retrieve operation of resources with dct:publisher being equal to dbpedia:TechCrunch, while Figure 8 illustrates the use of the KnowledgeStore Java client within an application for retrieving all the mentions of type nwr:entity_type_per.

3.2.2 SPARQL Endpoint

The SPARQL endpoint allows to query crystallized axioms in the entity layer using the SPARQL query language[27], a W3C standard for retrieving and manipulating data in Semantic Web repositories. This endpoint provide a flexible and Semantic Web-compliant way to query for entity data, and leverages the grounding of the KnowledgeStore data model in Knowledge Representation and Semantic Web best practices. Here below is the description of the `sparqlQuery()` operation offered by the SPARQL endpoint:[28]

- `sparqlQuery(query, dataset) : query solutions or RDF triples`]
  Evaluates the supplied SPARQL `query` on the RDF data encoding crystallized axioms or on a subset of it identified by the `dataset` parameter. The input *query* string could be in the SELECT, ASK, CONSTRUCT or DESCRIBE forms, while the optional *dataset* specification is a set of default graph URIs and named graph URIs (see FROM and FROM NAMED clauses of SPARQL). The expected output is either a list of *query solution* (tuples of variable bindings) for SELECT and ASK queries, or a set of RDF *triples* for CONSTRUCT or DESCRIBE queries

Figure 10 shows an example of querying some contextualized axioms stored in the KnowledgeStore, and the result obtained. On the left side, we have an excerpt of the

[27] `http://www.w3.org/wiki/SPARQL`

[28] The definition of the `sparqlQuery()` operation is based on the SPARQL protocol standard [Feigenbaum *et al.*, 2013]; indeed, the SPARQL protocol is used to implement this API operation.

```
import org.openrdf.model.*;
import eu.fbk.knowledgestore.*;
import eu.fbk.knowledgestore.model.*;

Store ks = new StoreClient("http://newsreader.fbk.eu/kstest");
Session s = ks.newSession("username", "password");

try {
    Cursor<Record> i = s.retrieve(KS.MENTION)
                        .where("nwr:entityType=nwr:entity-type-per")
                        .select(NIF.ANCHOR_OF, NWR.SYNTACTIC_HEAD)
                        .exec();
    while (true) {
        Record mention = i.next();
        if (mention == null) break; // cursor exhausted;

        String extent = mention.getUnique(NIF.ANCHOR_OF, String.class);
        String head = mention.getUnique(NWR.SYNTACTIC_HEAD, String.class);

        URI uri = myNEDSystem.disambiguate(head, extent);

        mention.set(KS.REFERS_TO, uri);

        s.merge(mention, MergeCriteria.override(KS.REFERS_TO));
    }
} finally {
    c.close();
}
```

based on Sesame API (http://www.openrdf.org)

XPath-based conditions

selection of output attributes

cursor model for streaming data

mentions & other objects are records of key-value pairs; URIs used as keys

merge criteria to combine new and old data

Figure 9: Using the KnowledgeStore client within a Java application.

Given this RDF data in the KS...

```
ex:module_01 {
  dbpedia:Volkswagen ex:marketShare "9.6%". }

ex:module_02 {
  dbpedia:Volkswagen ex:marketShare "12.3%". }

ckr:global {
  ex:ctx_15 a ckr:Context;
    ckr:hasModule nwr:module_01;
    sem:hasPointOfView ex:pov_19;
    sem:hasTimeValidity ex:time_2007.

  ex:ctx_16 a ckr:Context;
    ckr:hasModule ex:module_02;
    sem:hasPointOfView ex:pov_19;
    sem:hasTimeValidity ex:time_2011.

  ex:time_2007 a time:Interval;
    time:hasBeginning [
      time:inXSDDateTime "2007-01-01" ];
    time:hasEnd [
      time:inXSDDateTime "2007-12-31" ].

  ex:time_2011 a time:Interval;
    time:hasBeginning [
      time:inXSDDateTime "2011-01-01" ];
    time:hasEnd [
      time:inXSDDateTime "2011-12-31" ].

  ex:pov_19 a sem:PointOfView;
    sem:hasAuthority dbpedia:Forbes;
    sem:hasPointOfViewTime ex:pov_19_time.

  ex:pov_19_time a time:Instant;
    time:inXSDDateTime "2012-06-26".
}
```

... we ask for Volkswagen market share trend ...

```
SELECT ?share ?from ?to ?authority
WHERE {
  GRAPH ?m {
    dbpedia:Volkswagen ex:marketShare ?share
  }

  GRAPH nwr:global {
    ?ctx a ckr:Context;
      ckr:hasModule ?m;
      sem:hasPointOfView ?pov;
      sem:hasTimeValidity ?interval.

    ?pov a sem:PointOfView
      sem:hasAuthority ?authority.

    ?interval a time:Interval;
      time:hasBeginning ?from
      time:hasEnd ?to.

    ?begin time:inXSDDateTime ?start.

    ?end time:hasEnd ?end.
  }
}
```

... getting the following results:

share	from	to	authority
9.6%	2007-01-01	2007-12-31	dbpedia:Forbes
12.3%	2011-01-01	2011-12-31	dbpedia:Forbes

Figure 10: SPARQL endpoint example.

KnowledgeStore content showing the information on the market share of Volkswagen in two different contexts, one referring to 2007 and one to 2011, both having Forbes as associated authority. In our approach (see Section 2.1), each axiom corresponds to a set of ⟨subject, predicate, object⟩ triples within a named graph [Carroll *et al.*, 2005]—e.g., nwr:module_01 and nwr:module_02—that is linked to the context where the axiom holds—e.g., nwr:ctx_15 and nwr:ctx_16, which are the contexts associated to the axioms. On the right side we have (top box) a SPARQL query asking any market share content related to Volkswagen, the time validity of the information, and the authority that expressed it. As shown by the query, clients interacting with the SPARQL endpoint have to be aware of the contextual organization of data in the KnowledgeStore to properly formulate the query and interpret its results, that for the example are shown on the right side, bottom box.

Similarly to the CRUD one, the SPARQL endpoint is made available to the external KnowledgeStore users in two modalities: through an HTTP Server compliant to the SPARQL protocol, and as part of the Java client.

4 The KnowledgeStore Architecture and Implementation

Changes wrt the KnowledgeStore Architecture described Deliverable D6.1

- revision of system architecture, adapted to changes in data model (esp. axiom representation) and interfaces (esp. transactional guarantees of API operations) (Section 4.1)
- additional details about the use of HBase, including server-side filtering and integration with the OMID transaction manager (Section 4.1.1)
- additional details about the use of Virtuoso, including non-transactional updates and failure recovery (Section 4.1.2)
- description of the Java implementation of the system, featuring modular code organization and comprehensive manually and automatically generated documentation available online (Section 4.2)

This section describes the architecture of the KnowledgeStore and its software implementation. The KnowledgeStore is a client-server system that relies on distributed and scalable software components to store information of the data model and expose it through the CRUD and SPARQL endpoints. Section 4.1 describes the architecture of the system focusing on the main software components, namely *Hadoop and HBase*, the *Virtuoso triple store* and the KnowledgeStore *Frontend Server* that has been specifically developed to realize the KnowledgeStore functionalities on top of the other components. Section 4.2 provides an high level overview of the software implementation of the KnowledgeStore and, particularly, of the KnowledgeStore Frontend Server; additional details on the software implementation, including Javadoc documentation and auto-generated reports on various aspects of the code, are available online on the KnowledgeStore site.[29]

4.1 Architecture

As introduced in Section 1 with Figure 2, the KnowledgeStore is a storage server: the other NewsReader modules are KnowledgeStore clients that utilize the services it exposes to store and retrieve all the shared contents they need and produce. Figure 11 shows the overall KnowledgeStore architecture, highlighting its client-server nature.

Client side The client side (upper part of Figure 11) consists of a number of applications that access the KnowledgeStore through its two CRUD and SPARQL endpoints, either by direct HTTP interaction (for applications in any programming language), using the specifically developed Java client (for Java applications) or any of the available SPARQL client libraries[30] for accessing the SPARQL endpoint, thanks to its standard-based nature. From a functional point of view, client application may carry out different tasks:

[29] `http://newsreader.fbk.eu/knowledgestore`

[30] See `http://www.w3.org/wiki/SparqlImplementations`.

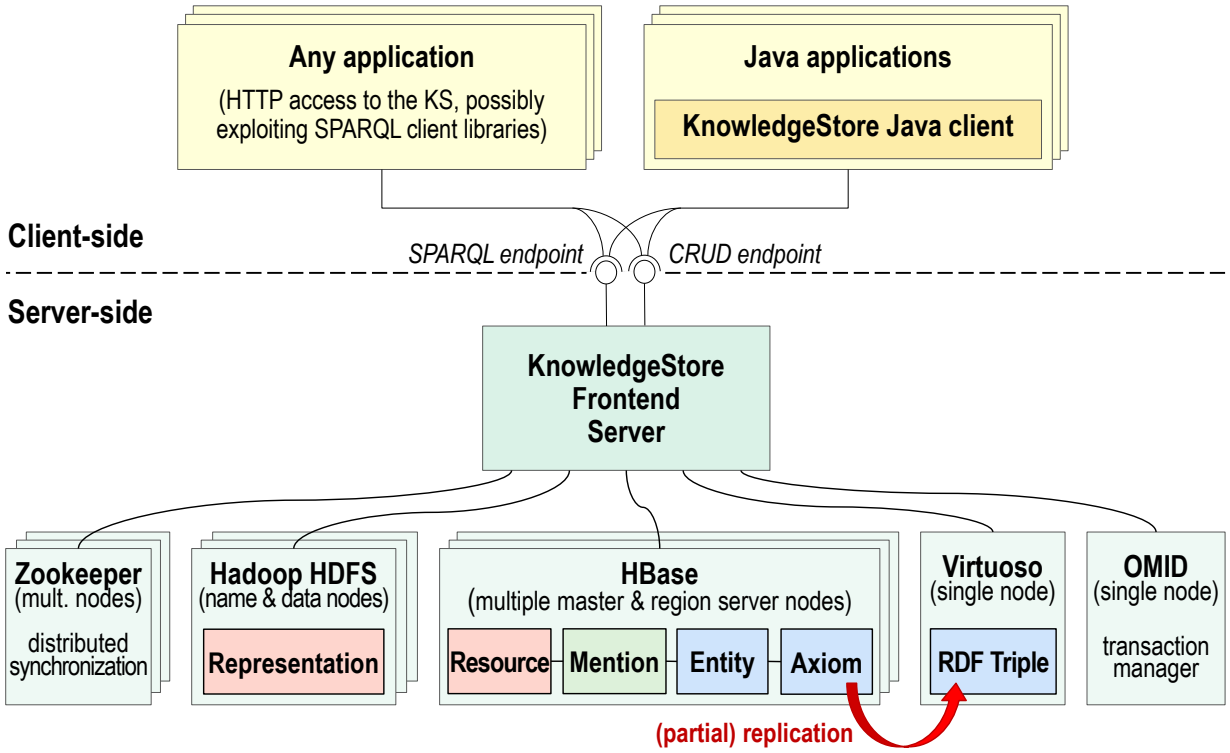

Figure 11: KnowledgeStore architecture.

- *populators* are clients whose main purpose is to feed the KnowledgeStore with new data; they play an important role in the NewsReader system, since they write into the KnowledgeStore the basic contents needed by other applications, such as the resources supplied by data providers and the background knowledge about entities;
- *linguistic processors* can also act as clients, by reading their input data from the KnowledgeStore and writing back the results of their computation;
- other client applications may be mainly interested in reading data from the KnowledgeStore: an example is the *Decision Support Tool Suite* of WP7.

Server side The server side part of the architecture (lower part of Figure 11) consists of a number of software components distributed on a cluster of machines that are accessed through a KnowledgeStore frontend server:

- the *Hadoop HDFS* filesystem provides a reliable and scalable storage for the physical files holding the representation of resources (e.g., texts and linguistic annotations of news articles);
- the *HBase* column-oriented store builds on the Hadoop filesystem to provide databases services for storing and querying semi-structured information about resources, mentions and entities;
- the *Virtuoso* triple store stores and indexes crystallized axioms to provide services supporting reasoning and online SPARQL query answering, which cannot be easily

and efficiently implemented in HBase or Hadoop;
- the *OMID* transaction manager[31] is used in combination with HBase to enforce the transactional guarantees of KnowledgeStore API operations (see Section 3.1);
- the *ZooKeeper* synchronization service is used to access and manage HBase nodes.
- the KnowledgeStore *frontend server* has been specifically developed to implement the operations of the two CRUD and SPARQL endpoints on top of the components listed above, handling global issues such as access control, data validation and operation transactionality.

Not shown in Figure 11 are the additional tools and scripts for managing the complexity of software deployment in a cluster environment (potentially a cloud environment); they include, for example, the management scripts for infrastructure (daemons) deployment, start-up & shut-down, data backup & restoration and gathering of statistics. It is worth noticing that the KnowledgeStore is a passive component, without any active role concerning the orchestration of other NewsReader modules. External orchestration—if needed—may be defined within WP2 in light of the general NewsReader system architecture; for instance, it might employ an external orchestrator polling (or being notified by) the KnowledgeStore about the availability of new contents, which may activate other processing modules.

In the following, we present the main server-side software components of the KnowledgeStore architecture, namely HBase & Hadoop (Section 4.1.1), Virtuoso (Section 4.1.2) and the KnowledgeStore Frontend Server (Section 4.1.3).

4.1.1 HBase & Hadoop

Hadoop[32] and HBase[33] are frameworks developed by Apache to manage scalability for file systems and databases, respectively. Distributed computation on multiple nodes, replication and fault tolerance with respect to single node failure are their key features. HBase is particular suited for random, real time read/write access to huge quantity of data (such as *big data*), when the data's nature does not require a relational model. HBase belongs to the *NoSQL* database family: it provides a mechanism for storage and retrieval of data that use looser consistency models than traditional relational databases in order to achieve horizontal scaling and higher availability. It does not (natively) support SQL-like queries.

The KnowledgeStore utilizes the Hadoop distributed file system (DFS) to store resource representations, that is the physical files such as news documents or custom annotations provided by the linguistic processors. HBase is used as a database to store the remaining information, with dedicated tables for storing resource metadata, mentions, contexts and entities with their metadata. For the table schema, a "blob approach" has been adopted for all the tables. In this approach each object is stored in a single row with a single column entry that encodes all the attributes and related values associated to such object.

[31] `https://github.com/yahoo/omid`
[32] `http://hadoop.apache.org`
[33] `http://hbase.apache.org`

The encoding is based on schemas compliant with the *Apache Avro*[34] data serialization system. Benefits of this solution include space efficiency and transactional update of object values, as single-row operations are inherently transactional in HBase. Operations of the KnowledgeStore API may however affect multiple rows in different tables for each modified object, as happen, for instance, when a new mention is stored and the rows for its containing resource and associated entity must be modified to link them to the mention. To provide the transactional guarantees of the KnowledgeStore API for these operations in presence of multiple concurrent clients we used the OMID transaction package[35], which provides a full transaction manager over HBase. OMID exploits the versioning capabilities of HBase to realize a Multiversion Concurrency Control (MVCC) mechanism[36] on top of HBase, similarly to many databases. Transactionality of a read-only operation is achieved by reading the snapshot of data produced by the most-recently completed read-write operation. Transactionality of a read-write operation is achieved by storing modified data with an incremented version number, while preserving old data; when the operation completes, possible conflicts due to the concurrent modification of the same object by other operations are detected by OMID, and resolved by allowing only one of these operations to succeed and persistently store its data.

The storage of data of the entity layer in HBase deserves a special description, as this data is also (partially) stored in the triple store. Figure 12 shows an example of how this data is stored in the two systems. Within HBase, the entity URI, URIs of referring mentions and the axioms describing an entity with their metadata are all stored in an *entity* table; context definitions are instead stored in a *context* table whose rows are referred by axioms of the entity table. This organization represents a change with respect to the design of deliverable D6.1, which provided for an axiom and a context tables, and allows to lookup the description of an entity in a single, more efficient operation. Figure 12 shows also how entities can be both ABox instances (dbpedia:Volkswagen) and TBox concepts (dbo:Company). It also shows that axiom metadata (e.g., provenance and confidence values) is only stored within HBase, as (i) it is often irrelevant to SPARQL user queries, and (ii) it would cause an explosion of the number of triples stored in the triple store, causing a severe degradation of performances.[37]

4.1.2 Virtuoso

In order to support SPARQL queries on entity data received via the KnowledgeStore SPARQL endpoint (see Section 3.2.2), axioms are indexed in a triple store by storing using the RDF representation described in Section 2.1, i.e., as sets of ⟨subject, predicate, object⟩ RDF triples within named graphs (the *modules*) that are connected to context

[34] `http://avro.apache.org/`

[35] `https://github.com/yahoo/omid/wiki`

[36] `http://en.wikipedia.org/wiki/Multiversion_concurrency_control`

[37] Note, however, that in the KnowledgeStore implementation it is always possible to go back and forth from one representation to the other, since axioms are uniquely identified by their ⟨subject, predicate, object, context⟩ components which are stored both in HBase and in the triple store.

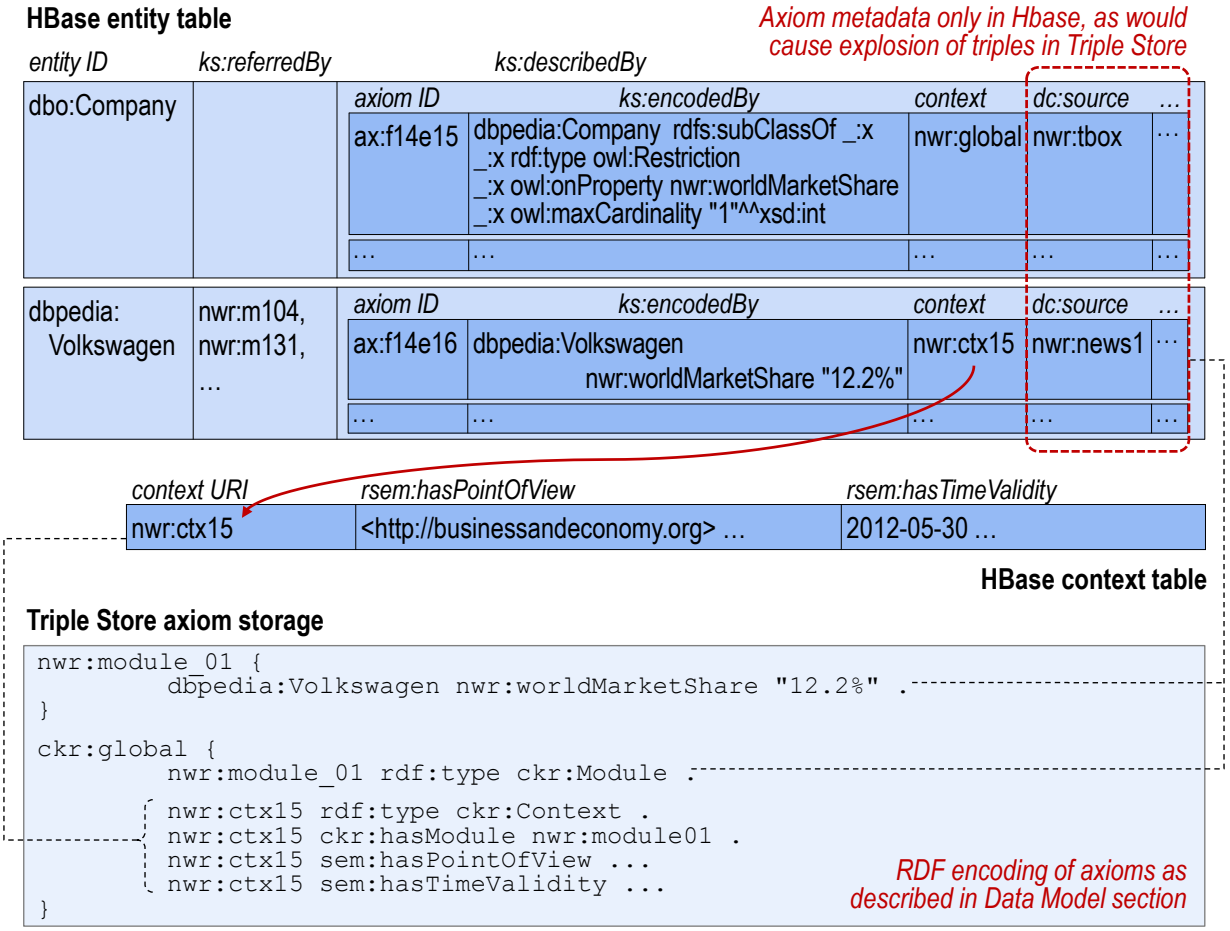

Figure 12: Axiom representation in HBase and in the Virtuoso Triple Store.

definitions in a specific ckr:global graph; as previously anticipated and shown in Figure 12, axiom metadata is not indexed and cannot thus be directly queried using SPARQL. The Open Source Edition of the Virtuoso triple store[38], version 7.0.0, has been chosen, motivated by its excellent performances in recent (April 2013) benchmarks[39] and its GPL v2 license. The Open Source Edition is limited to a single node deploy; additional scalability and transparent fault tolerance can be obtained using the (commercial) Enterprise Edition.

Virtuoso is accessed by the KnowledgeStore Frontend Server via the OpenRDF Sesame API[40], using the Virtuoso Sesame driver.[41] The Sesame API enables a uniform access to triple stores from Java applications, thus making easier to replace Virtuoso with a different triple store, should the need arise within or beyond NewsReader (e.g., for scaling up but

[38] http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/

[39] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V7/

[40] http://www.openrdf.org/

[41] We customized the Virtuoso Sesame driver to improve bulk loading performances when RDF triples are organized in many named graphs; based on the results we will measure, modifications will be possibly released to the Virtuoso community.

```
?ctx { ?x rdf:type ?c1 }
ks:global { ?c1 rdfs:subClassOf ?c2 }
-----------------------------------
?ctx { ?x rdf:type ?c2 }
```

(a) Contextual version of RDFS9

```
?ctx1 { ?s ?p ?o }
ks:global { ?ctx2 skos:broader ?ctx1 }
-----------------------------------
?ctx2 { ?s ?p ?o }
```

(b) Propagation from broader contexts

Figure 13: Examples of inference rules

also for scaling down the system by adopting a more lightweight triple store). Although the Sesame API allows for a transactional access to triple stores, performances of transactional data ingestion into Virtuoso resulted inadequate to the needs of the KnowledgeStore. Therefore, we decided to use Virtuoso exclusively in a non-transactional mode, adopting an approach that guarantees users of the SPARQL endpoint to access data that is always consistent and synchronized with the content stored in HBase and accessible via the CRUD endpoint. More in details, we consider content in HBase the *master copy* of data in the KnowledgeStore, relying on the fault-tolerance of HBase and the transactional data manipulation provided by OMID. Virtuoso is considered just an auxiliary index used exclusively for SPARQL queries. Synchronization of axiom data from HBase to Virtuoso is performed each time a data modification request to the KnowledgeStore API completes successfully, by excluding concurrent SPARQL accesses to Virtuoso (a simple multiple readers / single writer locking mechanism is used[42]). A synchronization failure (e.g., due to a problem with Virtuoso) is detected externally and, lacking a transactional log, triggers a full repopulation of Virtuoso starting from contents in HBase.[43]

The Virtuoso triple store component is tightly related to the support of *logical inference* in the KnowledgeStore. Inference aims at deriving the additional statements implied by stored data (ABox) and the ontologies defining its schema (TBox), and making them available as possible answers to applications and users queries. For instance, if a statement describes dbpedia:Volkswagen as a nwr:PublicCompany and nwr:PublicCompany is a subclass of nwr:Company in the KnowledgeStore background knowledge, then a query for all companies (e.g., from the decision support suite) is expected to return dbpedia:Volkswagen as an answer. Although logical inference is a task for the second year of the project (T6.3, starting month 15), it is worth noticing here that inference techniques such as closure materialization and rule-based reasoning can be efficiently implemented in a triple store such as Virtuoso, possibly on top of its SPARQL query answering capabilities[44]. Closure materialization may help to cope with the large amount of entity data stored in the KnowledgeStore, by storing the logical closure of loaded data thus speeding up online query answering. Cus-

[42] `http://en.wikipedia.org/wiki/Readers-writer_lock`

[43] The worst-case scenario repopulation is an expensive operation that may prevent SPARQL accesses for a long time (in the order of hours); therefore, this mechanism might be further refined in future releases of the KnowledgeStore, e.g., by repopulating from disk backups or using multiple instances of Virtuoso, one of which being always available for query answering while the others are synchronized.

[44] Rule-based reasoning can be implemented through the fix-point evaluation of SPARQL queries.

tomized rule-based reasoning can be necessary to consider the contextual validity of stored axioms, as no standardized ontological language currently supports reasoning with contextualized data. As a reference, Figure 13 shows two examples of customized inference rules: rule in Figure 13a extends rule RDFS9 (the rule responsible for the dbpedia:Volkswagen inference example above) and is applied on a per-context basis using TBox definitions (the rdfs:subClassOf triples) declared in a global context ks:global; rule in Figure 13b propagates statement holding in a context (e.g., time validity 2013) to other contexts declared (or found, via inference) to be narrower in scope (e.g., time validity 2013/12/15).

4.1.3 Frontend Server

The Frontend Server is a specifically developed Java daemon that provides the external API of the KnowledgeStore, implementing it on top of Hadoop, HBase and Virtuoso.

The implementation of the SPARQL endpoint is based on the SPARQL protocol[45] standardized by W3C. The CRUD endpoint is instead implemented as an HTTP ReST service using JSON for Linked Data (JSON-LD)[46] as the data format. JSON-LD is a W3C proposed recommendation for encoding Linked Data in JSON, thus inheriting the tool support, readability characteristics and developer friendliness of the JSON format while being a concrete RDF syntax at the same time. The adoption of JSON-LD greatly improves the usability of the CRUD endpoint, allowing both RDF-aware as well as JSON-based applications (even dynamic web sites using Javascript / AJAX) to easily interact with the KnowledgeStore. HTTP authentication is used to implement the security requirements of the API, while HTTP compression supports the efficient transmission of JSON-LD data.

Internally, calls to the SPARQL endpoint are all forwarded to Virtuoso, while the majority of calls to the CRUD endpoint are forwarded to HBase & Hadoop, although count() and retrieve() operations for axioms and entities without axiom metadata may be also answered by Virtuoso. Data modification operations are implemented by performing a number of transactions (one per affected object or group of objects) on HBase, using the OMID transaction manager. Upon successful completion of transactions, data modified in HBase is synchronized to Virtuoso; in the future, this will also trigger inference, which is transparently performed each time data is written through the API.

4.2 Implementation

The implementation of the KnowledgeStore architecture described in Section 4.1 comprises two activities: (i) development of the software components specific to the KnowledgeStore, namely the KnowledgeStore Frontend Server and the Java Client library; and (ii) setup of the test and production deployment environments where all the server components of the system are integrated in a unifying framework. Software development is detailed in Section 4.2.1, while setup of deployment environments is described in Section 4.2.2. Note that other software tools were specifically developed for the population of the KnowledgeStore

[45] http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/
[46] http://json-ld.org/

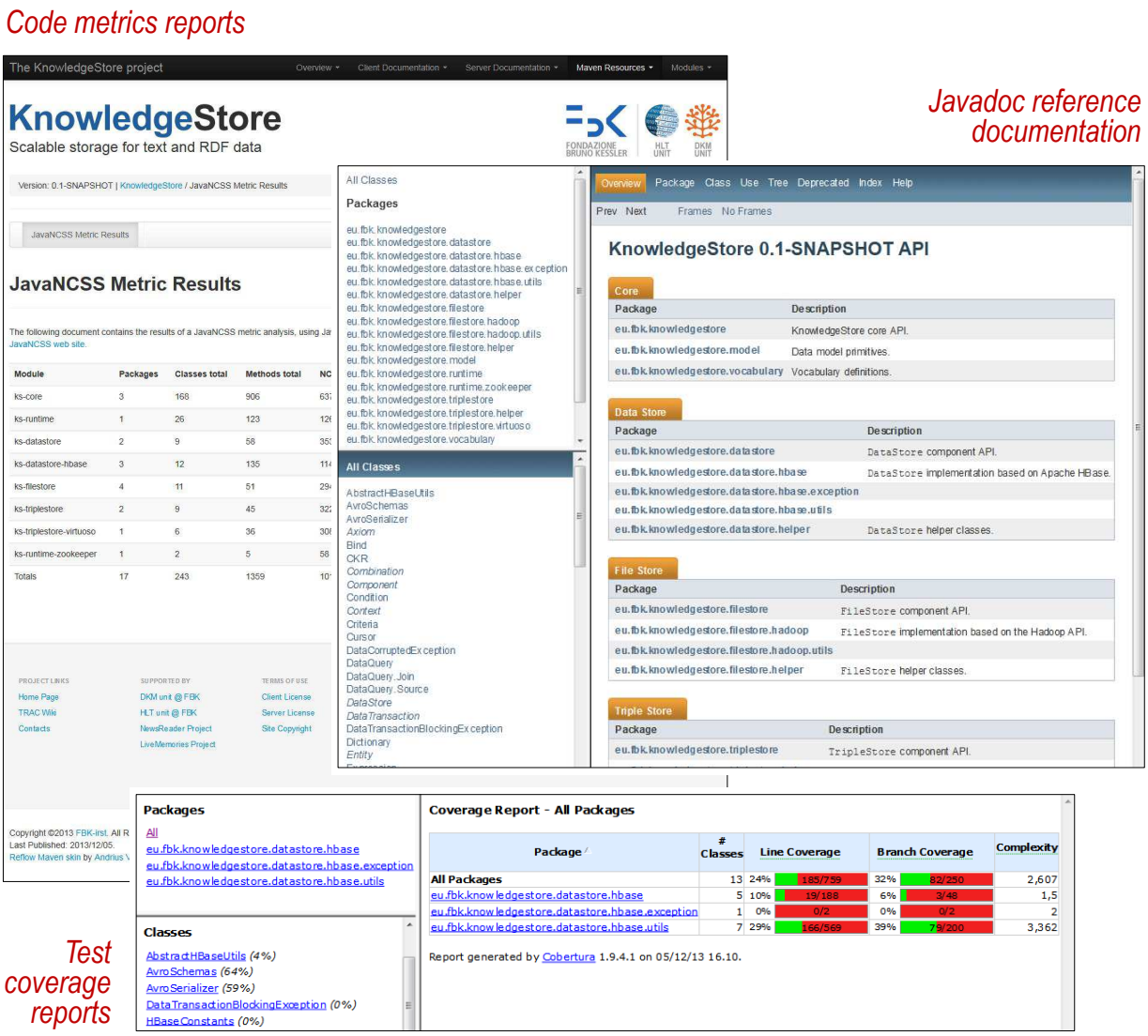

Figure 14: Examples of generated reports on the KnowledgeStore web site.

and the collection of background knowledge; since these tools operate as applications built on top of the KnowledgeStore, they are not described here but in Section 5.

4.2.1 Software development

The KnowledgeStore Frontend Server and the Java Client library have been developed in Java 1.6 following best practices for Java development.

The Apache Maven[47] build system and model have been used to manage the overall source code organization and all the phases of the build lifecycle (compiling, testing, re-

[47] http://maven.apache.org/

lease, ...), in combination with the Eclipse[48] Integrated Development Environment (IDE) for code writing. Maven represents the de-facto standard for Java software development. It eases the understanding and sharing of a software project among developers by favouring code modularity and convention over configuration. It provides a declarative dependencies model that facilitates building complex systems with many third-party libraries (as the KnowledgeStore), as well as using the components built in other applications. Finally, it supports the generation of comprehensive reports and Web documentation that provide at any moment a clear picture of the "health status" of a software project.

Maven capabilities have been fully exploited for the development of the KnowledgeStore. The adopted Maven setup allows for the automatic building, testing, packaging and distribution of the Frontend Server and the Java Client, with binaries of both components published online[49] according to Maven standards and easily importable in client application via the Maven dependency mechanism. The automatic generation of the project Web site has been configured, integrating both reports automatically generated by Maven and documentation manually authored that cover the deployment of the system and the use of the Java Client; examples of generated reports (including Javadocs) are shown in Figure 14. A Maven *multi-module* project organization has been adopted, with code organized in modules according to a functional criterion, as shown in Figure 15. This organization makes developing the different parts of the system easier, as work on each module can largely proceed independently of other modules, as well as more flexible, as new modules can be added and existing modules can be reimplemented in the future without breaking the overall structure. Following, a short description of the modules is reported:

ks-core Contains core abstractions and basic functionalities shared by the Frontend Server and the Java Client, defining a Java version of the KnowledgeStore API.

ks-runtime Contains general-purpose code used by different modules of the Frondend Server (e.g., configuration, synchronization, locking, file system access).

ks-filestore Realizes a *file store* sub-component that manages the files containing representations of resources (news, NLP annotations). It implements the standard read, write and delete operations over files on top of Apache Hadoop HDFS version 1.0.4, exploiting the scalability and fault tolerance features that Hadoop provides.

ks-datastore & ks-datastore-hbase Realize a *data store* sub-component managing semi-structured data about resources, mentions and entities. Module ks-datastore contains the abstract data store definition, while ks-datastore-hbase provides a concrete implementation on top of Apache HBase version 0.94.10, OMID and Apache Avro version 1.5.3; other implementation modules supporting alternative backends may be added later.

[48] `http://www.eclipse.org/`
[49] `http://newsreader.fbk.eu/knowledgestore`

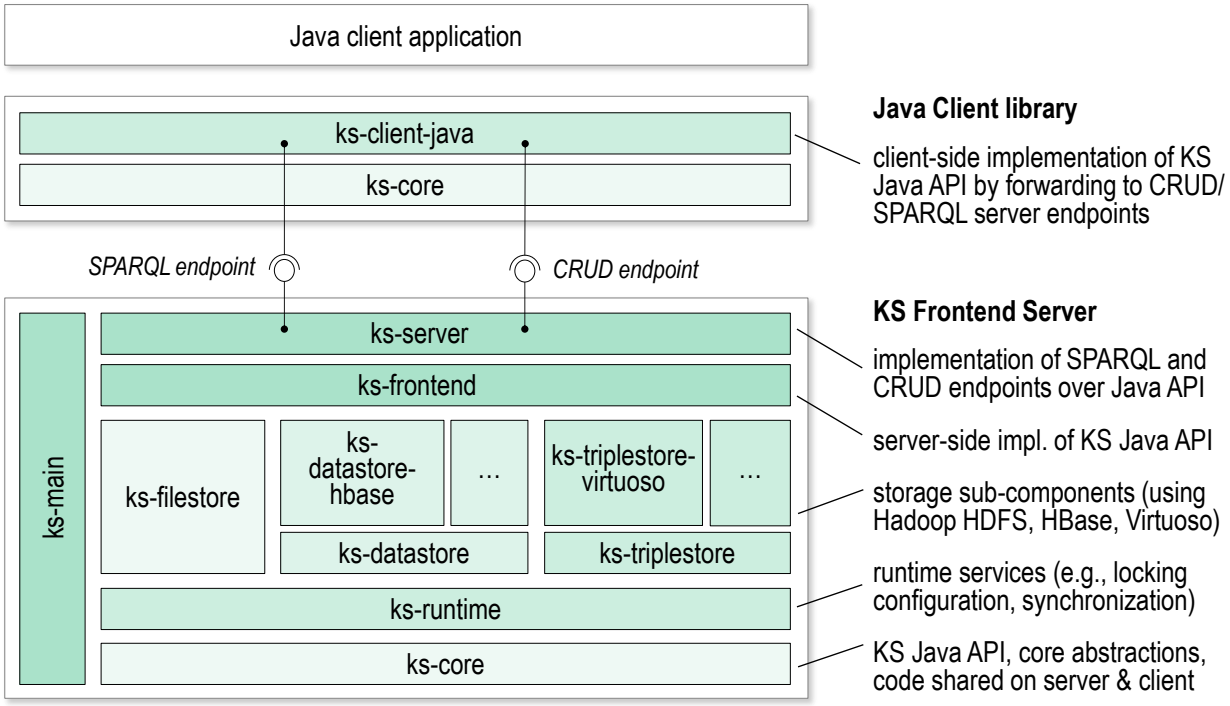

Figure 15: Modular code organization.

ks-triplestore & ks-triplestore-virtuoso Realizes a *triple store* sub-component for storing the RDF statements of axioms and supporting SPARQL querying. Module ks-triplestore contains the abstract definition of the sub-component, while ks-triplestore-virtuoso provides its implementation on top of Virtuoso version 7.0.0; other implementation modules for alternative backends may be added in the future.

ks-frontend Represents the core of the Frontend Server, implementing the Java version of the KnowledgeStore API on top of the file store, data store and triple store sub-components. This module provides a fully operational, non client-server version of the KnowledgeStore that can be embedded in applications similarly to an embedded database.

ks-server Implements the CRUD and SPARQL KnowledgeStore endpoints as HTTP ReST services on top of the ks-frontend module, enabling a client-server use of the system.

ks-main Implements the KnowledgeStore executable server daemon, by configuring and controlling the services provided by ks-server, ks-frontend and its sub-components.

ks-client-java Provides the Java Client library, building on top of the abstractions of ks-core and implementing the Java version of the KnowledgeStore API by translating API calls in HTTP requests to the CRUD and SPARQL server endpoints.

4.2.2 Deployment environments

To develop, test and operate the KnowledgeStore we have setup two kinds of deployment environments: (i) a single-machine setup and a (ii) a small cluster of four workstations. The former has been created for local development and fast testing; it integrates all the software components required by the KnowledgeStore server ready for use and is distributed among developers in the form of a VirtualBox[50] virtual machine. The latter is being used for distributed testing and the initial deployment of the KnowledgeStore. The workstations are commodity hardware with RAM ranging from 8 to 32 Gb and local disk size of 1 Tb, running Linux Red Hat Enterprise release 6.5. For both the environments, a number of scripts has been developed for managing the configuration, startup and shutdown of the system.

[50] `https://www.virtualbox.org/`

5 The KnowledgeStore Population

This section is about the population of the KnowledgeStore with resource, mention and entity data produced within the NewsReader project.

Resource and mention data come from the NLP pipeline of WP4 and is expressed according to the NewsReader Annotation Format (NAF). Storing this data in the KnowledgeStore implies parsing the NAF contents, extracting the contained resources and mentions and loading them in the system via the CRUD endpoint. These activities are specifically supported in WP6 with the realization of a *NAF populator*, described in Section 5.1, that acts as a bridge between the KnowledgeStore and the NLP pipeline of WP4.

Entity data, on the other hand, consists of RDF graphs containing either the *background knowledge* collected from external sources or the results of the NLP processing carried out in WP5. The population of the KnowledgeStore with this data is supported in WP6 with the realization of a general purpose, context and metadata-aware *RDF populator*, described in Section 5.2, and with the acquisition of background knowledge from Linked Open Data (LOD) sources, described in Section 5.3.

5.1 NAF populator

Starting from news documents, the linguistic processors of the NLP pipeline produce annotations encoded according to the NewsReader Annotation Format (NAF). Annotations in NAF files are organized on different layers and may include (explicit or implicit) representations of mentions and entities: in order to be shared among the NewsReader modules, such objects need to be identified in the NAF files and stored in the KnowledgeStore. Moreover, the original news documents, as well as the NAF files themselves, represents useful Resources to be shared through the KnowledgeStore.

As shown in figure 16, the NAF populator is the module that takes in input a NAF file, identifies the relevant information it conveys in terms of resources, mentions and entities and stores them in the KnowledgeStore interacting with its APIs. It is worth noticing here that the NAF populator is not expected to add any information to those encoded in the NAF files. Its duty is to recognize the formats in which the objects relevant to the KnowledgeStore are encoded in the NAF files, and transform such objects into invocations to the KnowledgeStore APIs to store them explicitly. Operations that add information to NAF file contents – such as coreference or linking – are outside the tasks of the NAF populator. Another aspect related to the previous is the assumption that the NAF populator is not expected to check the semantic correctness of the information encoded in the NAF files: it stores in the KnowledgeStore any storable data it is able to find. We can think to the NAF populator as a tool that transfers objects from the NAF format to the KnowledgeStore Data Model through the KnowledgeStore APIs.

Given its task, the most important issue that the NAF populator should address is the mapping of the NAF representations into the KnowledgeStore data model (see section 2). This is crucial because the KnowledgeStore can store only objects that comply with the data model underlying it. Let us consider an example of a piece of NAF file:

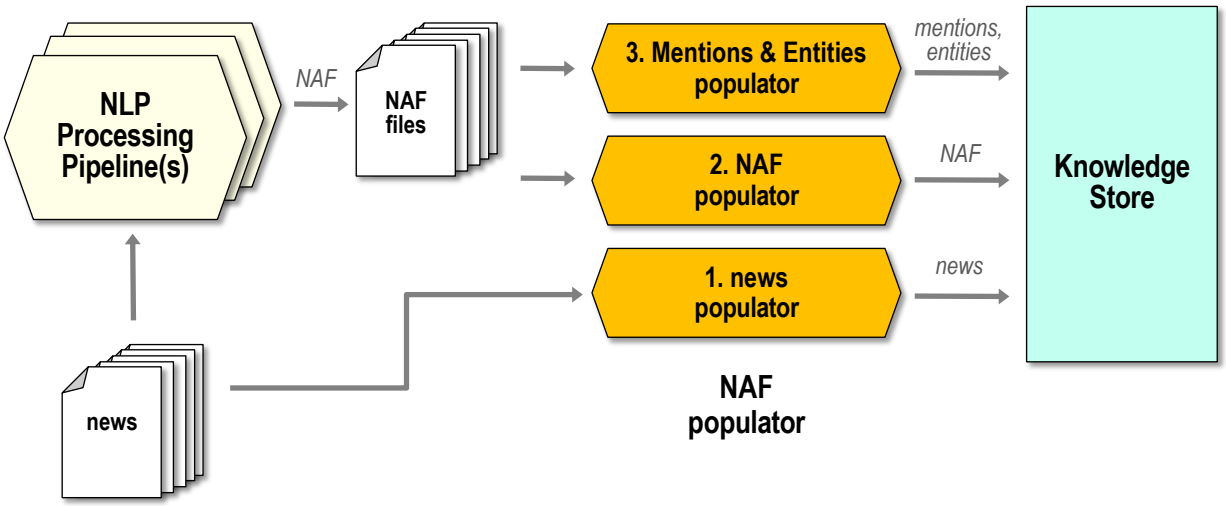

Figure 16: NAF population.

```
<text>
   <wf id="w1" length="5" offset="0" sent="1">Barak</wf>
   <wf id="w2" length="5" offset="6" sent="1">Obama</wf>
</text>
<entity id="e1" type="person">
   <references>
      <span>
         <word id="w1"/>
         <word id="w2"/>
      </span>
   </references>
</entity>
```

This portion of NAF encodes a mention whose main attributes are its type ("person") and its extent (the text "Barak Obama"). Other information related to this mention can be extracted, for example its starting character index in the text (being 0) and its ending character index (11). In order to be compliant with the KnowledgeStore data model, the populator should create a new mention with the following properties (in pseudo-code):

```
Mention m = new Mention();
m.set(nwr:entityType, nwr:entityTypePer);
m.set(nif:anchorOf, "Barak Obama");
m.set(nif:beginIndex, 0);
m.set(nif:endIndex, 11);
m.set(ks:containedIn, newsDocumentIdentifier)
```

The last line of code establishes a relation between the mention and the news document from which it has been extracted.

At this point one may suppose that the new mention is ready to be stored into the KnowledgeStore. That is not completely true, actually, because the mention lacks an identifier. Concerning identifiers, the KnowledgeStore assumes that resources, mentions and entities must be provided with their own identifiers, while for axioms and contexts a new identifier is automatically generated by the KnowledgeStore. Therefore the NAF populator has to find or assign a proper identifier for each new resource, mention or entity before storing them in the KnowledgeStore. For resources, the identifier is based on the value of the attribute nafPublicId contained in the NAF file, attribute that it is assumed to be uniquely generated. The identifier depends on the type of the resource and it is obtained as follows:

- for a news document, identifier is the string `$PREFIX + "news/" + $nafPublicId`
- for a NAF file, identifier is the string `$PREFIX + "naf/" + $nafPublicId`

where `PREFIX` is a URI such as `http://www.newreader-project.eu/`. The identifier of a mention is assigned on the basis of the position of its extent in the original news document, following the guidelines of the RFC 5147 IETF standard[51]. So the identifier of the mention described in the example above is the string `$PREFIX + "news/" + $nafPublicId + "#char=0,11"`.

The processing of a single NAF file is the basic functionality of the NAF populator and it is the building block for more complex operations along two dimensions: (1) the quantity and (2) the time. The former is an issue for the initial population, when the KnowledgeStore is empty and a very large number of NAF files are ready for a massive population operation. In this case different strategies can be developed to maximize the throughput of data exchange with the KnowledgeStore and the population speed. The latter concerns the different situations in which the NAF populator may operate after performing the initial population: for example new NAF files may be generated by the NLP pipeline according to the availability of additional news documents. This may happen weekly, daily or even more times per day. The frequency on which the NAF populator is activated and by which module, as well as a mechanism to notify when new data are available, has to be defined within the overall NewsReader system architecture.

5.2 RDF populator

The RDF populator (sources and binaries available online[52]) takes one or more RDF files in input, extracts the contained axioms together with their metadata (e.g., provenance) and contextual information and stores them in a running KnowledgeStore instance. Input axiom data must be represented as specified in Section 2.1, that is:

[51] `http://tools.ietf.org/html/rfc5147`
[52] `http://newsreader.fbk.eu/knowledgestore`

- the triples encoding an axiom must be stored in a named graph called *module*, which can host triples of multiple axioms sharing the same context and metadata;
- within a special ckr:global graph, the module URI is the subject of metadata triples that apply to all the axioms in the module;
- contexts must be defined in ckr:global and linked to axioms modules via ckr:hasModule;
- the RDF representation of structured values (e.g., OWL Time intervals, SEM point of views) of contextual dimensions or metadata properties must be placed in ckr:global.

Blank nodes are unsupported in the KnowledgeStore: if present in input, the RDF populator automatically replaces them with URIs via *skolemization*, assuming a file-based blank node scope[53]. Contexts URIs in input data are also ignored, as the KnowledgeStore automatically generates them based on the values of contextual dimensions. More in details, the RDF populator accepts the following parameters:

- one or more RDF input files, supporting different RDF syntaxes and optional compression; RDF and compression formats are automatically detected based on the file name (e.g., "input.trig.gz" is parsed as a gzip-compressed TriG file);
- in alternative, RDF data can be read from standard input with explicit specification of the RDF and compression formats, easing the integration of the RDF populator in NLP pipelines that connects modules via standard output to standard input piping;
- the URL, username and password to access a running KnowledgeStore instance;
- the default metadata and context to attach to parsed axioms if missing in the input RDF (optional parameters, no metadata and global context used by default);
- the *merge criteria* (see Section 3.2.1) for merging axiom metadata with metadata already stored in the KnowledgeStore for the same axioms (optional parameter, *union* of old and new metadata stored by default);
- the error file where to store the RDF representation of axioms rejected by the KnowledgeStore; manual correction and upload of these axioms can be done later (optional parameter, display of brief error summary with no generation of error file by default);
- a URI to be used in place of ckr:global (optional parameter, default is ckr:global).

Technically, the RDF populator is realized as a cross-platform Java application with a command line interface. The tool preprocesses the RDF input by sorting its triples, placing metadata and context information just before the triples that encode axioms; in a second pass, sorted triples are just scanned and translated into axioms that are streamed to the KnowledgeStore, keeping track of whether they are successfully stored or not. Sorting is performed using the sort system utility[54], which performs in-memory sort and falls back

[53] In RDF, blank nodes (bnodes) act as existential variables that denote some entity transiently and locally (to a file, graph), whereas URIs are persistent, global identifiers. As a consequence, blank nodes cannot be used as entity identifiers in the KnowledgeStore, also because no retrieval by ID facility could be supported in that case (they are variables, not identifiers). Skolemization is the process that replaces existential variables with function symbols; in RDF, it is used to replace blank nodes with auto-generated URIs that gives a stable identity to the entities denoted with the blank node.

[54] sort from GNU Core Utilities is available on different platforms, including Windows.

to external disk sort when memory is not enough. This approach addresses the fact that the order of triples in an RDF file is not given, with triples of an axioms, its context and metadata possibly scattered throughout the file. The use of sorting avoids the need to fully load input files in memory, thus enabling the processing of huge RDF files (like the ones containing the background knowledge described in the next section).

5.3 Acquisition of LOD background knowledge

Background knowledge consists of terminological and assertional data that describes entities, events, their relations and structure, and that supports NLP processing and the use case applications. Within NewsReader, background knowledge is collected in WP4 through the conversion of company datasets and other structured sources to RDF (see Deliverable D4.3.1: Structured data to RDF), and within WP6 from Linked Open Data (LOD) sources. Collected knowledge is then stored in the KnowledgeStore (using the RDF populator) and used to support NLP processing in WP5 (e.g., for entity and event coreference), the knowledge crystallization task of WP6 and the decision support tool suite of WP7.

This section reports on the activities for the collection of background knowledge from LOD sources carried out in WP6. As a first step, described in Section 5.3.1, a subset of selected LOD datasets and terminological resources were chosen for importing background knowledge, based on a set of criteria. The next step, described in Section 5.3.2, consisted in assembling a processing pipeline to extract, combine and augment relevant data from selected datasets and produce a coherent dataset ready to be imported in the KnowledgeStore. Statistics about the resulting dataset are presented in Section 5.3.3.

5.3.1 Data selection

The Linked Open Data (LOD) cloud is a collection of machine-readable RDF data about entities in different domains (persons, organizations, places, ...) and consisting of over 31 billions of triples in ∼300 datasets interlinked with 504 millions of links[55] Although LOD data presents shallow structure and semantics, the wealth of information conveyed, and the fact that this information is constantly updated (mainly through community efforts), with no need of manual intervention of experts, make data in the LOD cloud particularly useful for NewsReader. However, importing all this data as background knowledge is technically unfeasible due to its huge size, and also because not all of this data is relevant or can be exploited by the NLP modules and use case applications of NewsReader. A selection of data is thus necessary, and can be done based on the following criteria:

- *Linkability.* It must be possible for every entity gathered from LOD data to be linked to mentions in a news. This means that the entity URI must be among the URIs that can be assigned by the named entity disambiguation tool used in the project, which is currently DBpedia Spotlight[56] (see Deliverable D4.2.1: Event detection, version

[55] Statistics from `http://lod-cloud.net/state/` as of September 2011 (last available). The current number of triples and owl:sameAs links can be considerably greater.

[56] `http://spotlight.dbpedia.org`

1); in alternative, the entity URI should be reachable through a chain of owl:sameAs links from a supported URI. In practice, this criterion limits the choice to DBpedia in various languages and to datasets directly or indirectly linked to DBpedia (being DBpedia the hub of the LOD cloud, there are many datasets linked to it).

- *Focus on real world entities data.* Collected data should consist of descriptions of real world entities that are as complete as possible. This rules out data about entities that denote dynamically computed information (e.g., the results of some Web service, such as FlickrWrappr[57] that provides on-the-fly listings of Flickr images about an entity) and metadata in general (e.g., the template or the Wikipedia page revisions used for extracting a certain DBpedia entity).

- *Domain relevance.* Selected data should be relevant in the domains of interest for NewsReader. This criterion supports the inclusion of cross-domain datasets and also geographical datasets, as geographic information is ubiquitous in many domains including the economical-financial ones of NewsReader. Inclusion of specialized, domain-specific datasets (e.g., MusicBrainz[58] for music data) is instead questionable at this stage of the project, although some of them can be considered later if deemed useful based on the results of the first news processing activities.

- *Quality over quantity.* As background knowledge is assumed to be true and can be possibly used as ground truth when training NLP modules, only high-quality data must be collected. This rules out datasets known to have data quality issues, and also suggest the selection of smaller but cleaner versions of a dataset where available.

- *Prefer standard vocabularies.* Collected data should be expressed according to standard, properly designed vocabularies (e.g., Dublin Core, the DBpedia OWL ontology) that ease data querying and consumption. Data expressed according to auto-generated vocabularies, often large and noisy, should be avoided (e.g., raw infobox data from DBpedia).

- *TBox inclusion.* TBox definitions should be included for every predicate and class referenced in collected data, so to enable reasoning, and must include mapping axioms that align those concepts to general vocabularies that can ease the querying of data.

Given the criteria above, the following open licensed and well interconnected datasets were considered as candidates for partial inclusion in collected background knowledge (size statistics refer to the versions of these dataset available as of 15/12/2013):

[57] `http://wifo5-03.informatik.uni-mannheim.de/flickrwrappr/`
[58] `http://musicbrainz.org/`

dataset	DBpedia (http://dbpedia.org/)
description	Cross-domain Dataset extracted automatically from the Wikipedia in different languages (mainly from infoboxes) and representing the hub of the LOD cloud. It aims at providing as much of factual knowledge in Wikipedia as possible. Raw infobox data is provided as well as data mapped to a manually crafted DBpedia OWL ontology. [Auer *et al.*, 2007]
availability	RDF dump files; some localizations provide SPARQL and dereferenceable URIs
size	4M entities (ABox instances only), 470M triples EN version; 12.6M entities, 1.98B triples all languages

dataset	Freebase (http://www.freebase.com/)
description	Cross-domain dataset containing community-contributed interlinked data, structured according to schema generated and edited by users and linked to DBpedia. Acquired by Google in July 2010 and used as a source for the Google Knowledge Graph launched in May 2012. Linked to DBpedia. [Bollacker *et al.*, 2008]
availability	single RDF dump file, dereferenceable URIs
size	43M entities, 2.4B facts (1.9B triples in the RDF dump)

dataset	YAGO2 (http://www.mpi-inf.mpg.de/yago-naga/yago/)
description	Cross-domain knowledge base automatically extracted from Wikipedia, WordNet and GeoNames. It features a rich type taxonomy (350K classes) and annotation of facts with confidence value, time and space validity. A 95% accuracy has been manually measured. Linked to DBpedia. [Hoffart *et al.*, 2013]
availability	RDF dump files, dereferenceable URIs
size	120M facts about 10M entities

dataset	GeoNames (http://www.geonames.org/)
description	Geographic database containing the most significant geographical features of Earth (e.g., countries, populated places) with georeferencing and containment relationships. Used as a hub for geographical data. Linked to DBpedia.
availability	RDF dump file, dereferenceable URIs
size	8.3M entities, 125M triples

dataset	LinkedGeoData (http://linkedgeodata.org)
description	Geographic RDF dataset automatically derived from OpenStreetMap and thus providing information about user-contributed points of interest (POIs) not covered by GeoNames. Linked to DBpedia and GeoNames. [Stadler *et al.*, 2012]
availability	RDF dump files, REsT API
size	1B nodes, 20B triples

Among these candidates, for this first extraction of background knowledge we restricted our focus to the DBpedia datasets in the four languages of the projects: English, Spanish, Italian and Dutch; we chose the DBpedia 3.9 datasets on the main DBpedia web site for English and Spanish, and the more complete datasets on localized DBpedia web sites (as of 01/11/2013) for Italian and Dutch. This dataset selection provides a comprehensive core of background knowledge that is well interconnected and, above all, presents an homogeneous schema that both eases data consumption by KnowledgeStore clients and also allows us to initially focus on the merging of ABox data without worrying about TBox heterogeneities. Starting from these four DBpedia datasets, overall amounting to over 824M triples in 184 (partially overlapping) dump files, we applied the criteria previously listed to narrow down the selection. As a result, the following parts of the four DBpedia datasets were selected:

- entity types and properties based on the FOAF and DBpedia OWL vocabularies (files instance_types, instance_types_heuristic, mappingbased_properties_cleaned and persondata, plus *airpedia*[59] data for IT DBpedia);
- entity names based on Wikipedia titles (file labels);
- entity types based on YAGO2 classes (files yago_types and yago_taxonomy);
- entity types based on UMBEL classes (file umbel_links);
- entity categorization based on Wikipedia categories (files articles_categories, category_labels and skos_categories);
- entity categorization based on Wordnet 2.0 synsets (file wordnet_links);
- geographic coordinates of location entities (file geo_coordinates);
- links to entity images in Wikipedia (file images), with removal of dc:rights copyright metadata (images are all open licensed);
- links to Wikipedia pages, home pages and other Web pages with additional entities information (files external_links, homepages, wikipedia_links); where defined, inverse foaf:isPrimaryTopicOf links from pages to entities were dropped;
- brief language-dependent textual description of entities (file short_abstracts);
- owl:sameAs links among URIs of DBpedia in different languages and among URIs and IRIs assigned to the same entity[60] (files interlanguage_links, iri_same_as_uri).

Based on the *TBox inclusion* criterion, the RDFS/OWL definitions of the following vocabularies were also selected: DBpedia 3.9 OWL ontology; Dublin Core elements (DC) and terms (DCTERMS); Friend of a Friend (FOAF) vocabulary; Simple Knowledge Organization System (SKOS) vocabulary; schema.org[61] and UMBEL[62] concept definitions; Bibliographic Ontology (BIBO); WGS84 and GeoRSS[63] vocabularies for geographic data.

[59] The *airpedia* project (`http://www.airpedia.org/`) augments DBpedia data with accurate type information extracted from Wikipedia pages with machine learning techniques [Palmero Aprosio *et al.*, 2013].

[60] The newer IRIs supports a broader set of characters and result more readable especially for non-English languages. As tools may still use URIs rather than IRIs (e.g., for linking a mention to an entity), we decided to include both kinds of identifiers interlinked with owl:sameAs links.

[61] `http://schema.org/docs/schemaorg.owl`

[62] `https://raw.github.com/structureddynamics/UMBEL/master/Ontology/umbel.n3` and file umbel_reference_concepts.n3 in GitHub repository.

[63] `http://www.w3.org/2005/Incubator/geo/XGR-geo/W3C_XGR_Geo_files/geo_2007.owl`

```
# prefix definitions omitted

SELECT ?name ?surname
WHERE {
  ?uri a dbo:Person;
    foaf:givenName ?name;
    foaf:familyName ?surname.
}
```

(a)

```
# prefix definitions omitted

SELECT ?name ?surname
WHERE {
  { ?uri1 a dbo:Person } UNION
  { ?uri1 a dbo:Artist } UNION
  ... for all dbo:Person subclasses ...
  { ? uri1 a dbo:Religious}
  ?uri2 foaf:givenName ?name.
  ?uri3 foaf:familyName ?surname.
  { ?uri1 owl:sameAs ?uri2. ?uri2 owl:sameAs ?uri3 } UNION
  { ?uri1 owl:sameAs ?uri2. ?uri3 owl:sameAs ?uri2 } UNION
  { ?uri1 owl:sameAs ?uri3. ?uri2 owl:sameAs ?uri3 } UNION
  { ?uri1 owl:sameAs ?uri3. ?uri3 owl:sameAs ?uri2 } UNION
  { ?uri2 owl:sameAs ?uri1. ?uri2 owl:sameAs ?uri3 } UNION
  { ?uri2 owl:sameAs ?uri1. ?uri3 owl:sameAs ?uri2 } UNION
  { ?uri3 owl:sameAs ?uri1. ?uri2 owl:sameAs ?uri3 } UNION
  { ?uri3 owl:sameAs ?uri1. ?uri3 owl:sameAs ?uri2 }
}
```

(b)

Figure 17: Example of SPARQL query with (a) and without (b) smushing and inference.

5.3.2 Data processing

Selected LOD files cannot be simply "concatenated" to produce the background knowledge dataset. Data must be filtered on a per-file basis, in order to remove unwanted data. Filtered data from different dataset must then be *smushed*[64], i.e. merged so that provenance metadata is preserved, duplicate triples are removed, and each entity identified by multiple URIs (connected by owl:sameAs links) is given a unique URIs that is used as the subject of triples describing the entity. Resulting data can then be augmented with (a computable subset as large as possible of) the inferred triples that derive from the included TBox definitions, before producing the final dataset.

Smushing and inference materialization are particularly important at this stage of the project, as they partially alleviate the problems caused by the lack of inference in the first version of the KnowledgeStore[65]. Without them, queries have to be written in a way that consider possible inferences and the effects of owl:sameAs triples if completeness of query results is desired[66], which leads to an explosion of a query complexity, as shown in Figure 17 for a simple query extracting names and surnames of persons in the background knowledge. In fact, smushing and inference materialization will also help when inference will be added to the KnowledgeStore, as they will reduce the load posed on the system for importing background knowledge.

In order to perform the required data filtering, smushing and inference materialization, a processing pipeline has been assembled that automatize these tasks based on a configuration file specifying the URLs of the files to process and how to process them. This pipeline represents an asset that will be exploited (via reconfiguration and possibly extension) later

[64] `http://patterns.dataincubator.org/book/smushing.html`

[65] According to the Project Description of Work, the development of the reasoning services on top of the KnowledgeStore is scheduled for M15.

[66] Note that completeness via query rewriting may be theoretically impossible depending on the logical fragment considered (e.g., RDFS, OWL 2 RL).

in the project to collect additional background knowledge from LOD sources. The pipeline is organized in four stages described next: *download*, *filtering*, *merging*, *analysis*.

Download stage Dataset and vocabulary files listed in the pipeline configuration are download from their source locations (if locally missing or newer), and trigger further processing in the next stages of the pipeline.

Filtering stage Each downloaded file is parsed, filtered and saved using a common format (TriG, as it supports named graphs) and compression method (gzip, due to a good tradeoff between compression ratio and speed). Filtering is performed in a single pass on a per-triple basis. It allows to drop triples with specific predicates and types configured on a per-file basis and, for every file, to remove literals not in a project language and to rewrite blank nodes making them globally unique (this avoid possible clashes when data from multiple files is merged). Triples in each filtered file are placed inside a named graph associated to the dataset, so to keep track of provenance in the following processing.

Merging stage This stage merges the filtered files previously generated, performing smushing and inference materialization and producing the final background knowledge file. Three passes are required to process and merge filtered files:

- The first pass extracts TBox definitions that are stored in a TBox output file. TBox definitions are identified by searching for triples having selected properties and classes from the RDF, RDFS and OWL vocabularies in predicate and object positions.

- The second pass scans filtered files for owl:sameAs links, which are used to build an in-memory "URI rewriting" data structure used later for assigning a unique, canonical URI to every entity. The size of the in-memory structure grows linearly with the number of distinct URIs linked by some owl:sameAs link; 60 bytes per URI has been measured on average, a number small enough to allow processing hundreds of millions of owl:sameAs links on a small workstation (16 to 32 GB memory).

- The third pass exploits the in-memory URL rewriting data structure and the TBox file, augmented with definitions inferred based on RDFS rules, to perform smushing and materialization of RDFS inferences at the ABox level, generating the resulting background knowledge file. ABox inference materialization is efficiently done on a per-triple level thanks to the restriction to RDFS (this would not be possible with OWL). Removal of duplicates, instead, requires to sort data, which is done using external merge-sort[67] due to the huge size of data involved (the sort utility is used).

[67] http://en.wikipedia.org/wiki/External_sorting

Table 1: Number of triples per source in produced dataset.

Source	rdf:type	owl:sameAs	ABox (other)	TBox	Total
DBpedia EN	110 270 306	21 219 244	96 317 680	450 970	228 258 200
DBpedia ES	7 001 275	11 865 956	19 298 294	0	38 165 525
DBpedia IT	8 488 011	11 287 650	18 017 981	0	37 793 642
DBpedia NL	5 354 073	11 315 456	12 707 068	0	29 376 597
misc. vocabularies	273 499	0	378 447	40 274	692 220
All sources	122 075 613	25 552 474	139 543 076	491 244	287 662 407

Analysis stage In this stage, a pass is done on the generated background knowledge file to collect a different number of statistics (see next section) and to generate a TBox file with concepts annotated with number of occurrences, which can later be imported in tools such as Protégé to better understand what is contained in collected background knowledge.

Technically, the pipeline is realized with a Java processing tool and a Python orchestration tool specifically developed for the purpose. Both tools will be released as open source resources together with the pipeline configuration file, thus making the whole process repeatable and reconfigurable by anyone. Using these tools and the current pipeline configuration on a RedHat 6.4 (Linux 2.6) workstation with an Intel(R) Core(TM) i7 CPU, 16 GB RAM and 500 GB disk, 73 files with 3.06 GB of compressed RDF data are downloaded, for a total of 309M triples; filtering is done in 38 minutes (136K triples/sec on average), returning 271M triples that are merged in 96 minutes (47K triples/sec); the result consists of 288M triples that are analyzed in 68 minutes (71K triples/sec).

5.3.3 Result statistics

The first release of LOD background knowledge, resulting from the selection and processing activities described above, is a dataset of 288M triples about 11.7M distinct entities.[68]

Tables 1 and 2 provide some statistics about the number of triples and entities in the produced dataset that were extracted from the different source datasets, that is DBpedia in the four project languages plus the additional imported TBox vocabularies (considered as a single dataset for simplicity). The numbers of triples per source dataset are reported in Table 1, divided among rdf:type triples, owl:sameAs triples, other ABox triples (essentially expressing entities properties) and TBox triples. The number of entities per source dataset are reported in Table 2, divided based on the types used in the annotation guidelines of WP3, i.e.: persons (PER), organizations (ORG), locations (LOC), products (PRO), financial entities (FIN) and events (EVENT), with OTHER representing entities that could

[68] Entities in the produced dataset have been counted by selecting distinct URIs appearing as the subject of some rdf:type statement and having a named OWL class as its object. This broad definition covers both ABox and TBox concepts, differently from the statistics provided by DBpedia that accounts only for ABox instances.

Table 2: Number of background knowledge entities per source in produced dataset.

Source	PER	ORG	LOC	PRO	FIN	EVENT	OTHER	All types
DBpedia EN	2 308 984	684 693	599 792	488 354	777	298 157	5 073 595	9 095 404
DBpedia ES	337 816	102 850	212 462	113 085	482	80 586	1 092 343	1 855 749
DBpedia IT	471 123	108 839	210 965	169 259	507	165 863	1 069 301	2 099 258
DBpedia NL	207 454	76 045	247 639	79 035	337	42 545	1 059 892	1 688 977
TBox vocabularies	0	0	0	0	0	0	27 134	27 134
All sources	2 803 131	791 620	704 221	623 845	895	382 134	6 724 209	11 682 908

not be classified under previous types. [69] In both tables, triples and entities originating from multiple source datasets are counted in each dataset, thus the values reported in "All sources" are not the sum of the values reported for the different sources. Similarly, the categorization among entity types is not exclusive in the data (although it should be), hence values in "All types" are not the sum of values reported for the different types.

Table 3 provides an overview of the contents of the produced background knowledge dataset, focusing on entity types and disregarding source datasets. The same entity types of Table 2 are used, further divided in entity sub-types (e.g., companies or political bodies for ORGs).[70] For each entity type and sub-type, the total numbers of distinct entities and associated triples are reported, with the average number of rdf:types triples, owl:sameAs triples, ABox property axioms and distinct ABox predicates per entity also reported (the higher the latter number, the more rich the description of an entity). Note that TBox triples are not considered in this table; moreover, an entity classified with multiple types is considered for each one of these types, hence the values reported in "All entities" are not the sum of the values reported for the different entity types.

Statistics per source and entity type, as well as detailed statistics about every TBox type and property that appears in the produced dataset have been computed as part of the analysis stage, and are distributed on the KnowledgeStore web site in the form of an annotated *statistics ontology*.[71] This ontology can be imported in tools for ontology editing and browsing such as Protégé, as shown in Figure 18, and can help in understanding and using the dataset, e.g., by supporting the construction of SPARQL queries.

[69] Classification according to the types of the annotation guidelines has been performed on the basis of the DBpedia types associated to entities using the following mapping: dbo:Person → PER; dbo:Organisation, dbo:Monastry → ORG; dbo:NaturalPlace, dbo:PopulatedPlace, dbo:ProtectedArea, dbo:SiteOfSpecialScientificInterest, dbo:WineRegion, dbo:FrenchSettlement, dbo:CelestialBody → LOC; dbo:Work, dbo:Database, dbo:Device, dbo:Drug, dbo:Flag, dbo:Food, dbo:Language, dbo:Aircraft, dbo:Automobile, dbo:Locomotive, dbo:Rocket, dbo:Ship, dbo:Train, dbo:SportFacility, dbo:ArchitecturalStructure, dbo:WorldHeritageSite, dbo:Monument, dbo:SkiArea → PRO; dbo:Currency → FIN; dbo:Event, dbo:Holiday, dbo:Award, dbo:Sales, dbo:SportsSeason → EVENT.

[70] Classification according to sub-types has been also performed based on DBpedia OWL types. The mapping is straightforward and is omitted for brevity reasons.

[71] The statistics ontology is distributed in two version: the full version with all concepts (15M triples) and a shrinked, more manageable version with concepts having more than 100 instances (100K triples).

Table 3: Number of entities, triples and properties per entity type in produced dataset.

Entity type	Total entities	Total triples	Avg. rdf:type	Avg. owl:sameAs	Avg. properties	Avg. predicates
PER	2 803 131	105 053 448	20.4	1.8	14.5	8.3
artists	404 177	19 100 547	25.5	2.8	18.2	8.9
military	30 796	2 780 554	49.6	3.3	36.4	20.1
nobles and monarchs	16 130	1 387 080	38.5	10.8	35.7	17.0
other professionals	24 992	1 758 313	42.5	3.5	23.7	12.0
politicians	54 119	4 048 227	40.7	4.2	29.0	15.7
religious	18 161	1 371 342	41.7	6.0	27.0	13.2
scientists	23 051	1 999 344	48.3	5.8	31.7	15.8
sportsmen	1 190 960	39 586 100	18.8	0.9	12.6	7.5
ORG	791 620	31 121 391	20.4	2.2	15.8	9.1
broadcasters	52 586	3 196 077	29.5	2.4	27.9	17.2
companies	125 375	4 212 340	15.1	2.8	14.9	8.0
educational	58 135	3 367 289	30.9	2.5	23.5	15.5
military	21 113	1 059 334	27.4	2.9	18.9	11.7
music bands	65 846	4 576 673	33.4	4.3	30.8	13.5
political bodies	15 369	549 236	11.6	4.7	18.6	10.1
religious	121	8580	17.3	19.7	33.5	9.2
sport	287 124	6 987 835	14.6	1.1	7.8	5.1
trade union	1489	82 705	36.1	1.9	16.6	9.9
LOC	704 221	36 963 630	18.8	6.9	25.9	15.3
geographical	122 413	5 895 993	21.7	6.0	19.5	12.3
geopolitical	623 307	33 999 783	19.1	7.2	27.2	16.0
PRO	623 845	28 995 543	16.9	2.9	25.8	14.9
art and entertainment	494 391	24 234 133	17.3	2.6	28.1	16.5
drugs	11 767	443 573	14.9	5.6	16.2	9.6
facilities	294 450	15 320 019	24.6	3.5	22.9	14.4
food	7042	209 544	8.9	4.1	15.8	8.6
ict products	37 040	2 046 616	22.1	4.6	27.7	13.8
transportation means	55 740	2 630 498	22.3	3.3	20.7	12.8
weapons	5372	273 936	23.3	6.1	20.7	11.7
FIN	895	51 581	22.6	13.9	20.3	7.0
EVENT	382 134	19 672 686	15.9	4.3	30.4	16.2
elections	6085	152 227	3.9	2.7	17.5	10.5
military conflicts	22 716	1 204 641	17.4	5.8	28.9	12.4
other entertainment	2274	133 388	30.3	3.6	23.8	11.4
sport	112 843	3 049 886	7.5	3.2	15.8	8.0
All entities	11 682 908	258 557 750	10.0	1.8	9.9	5.9

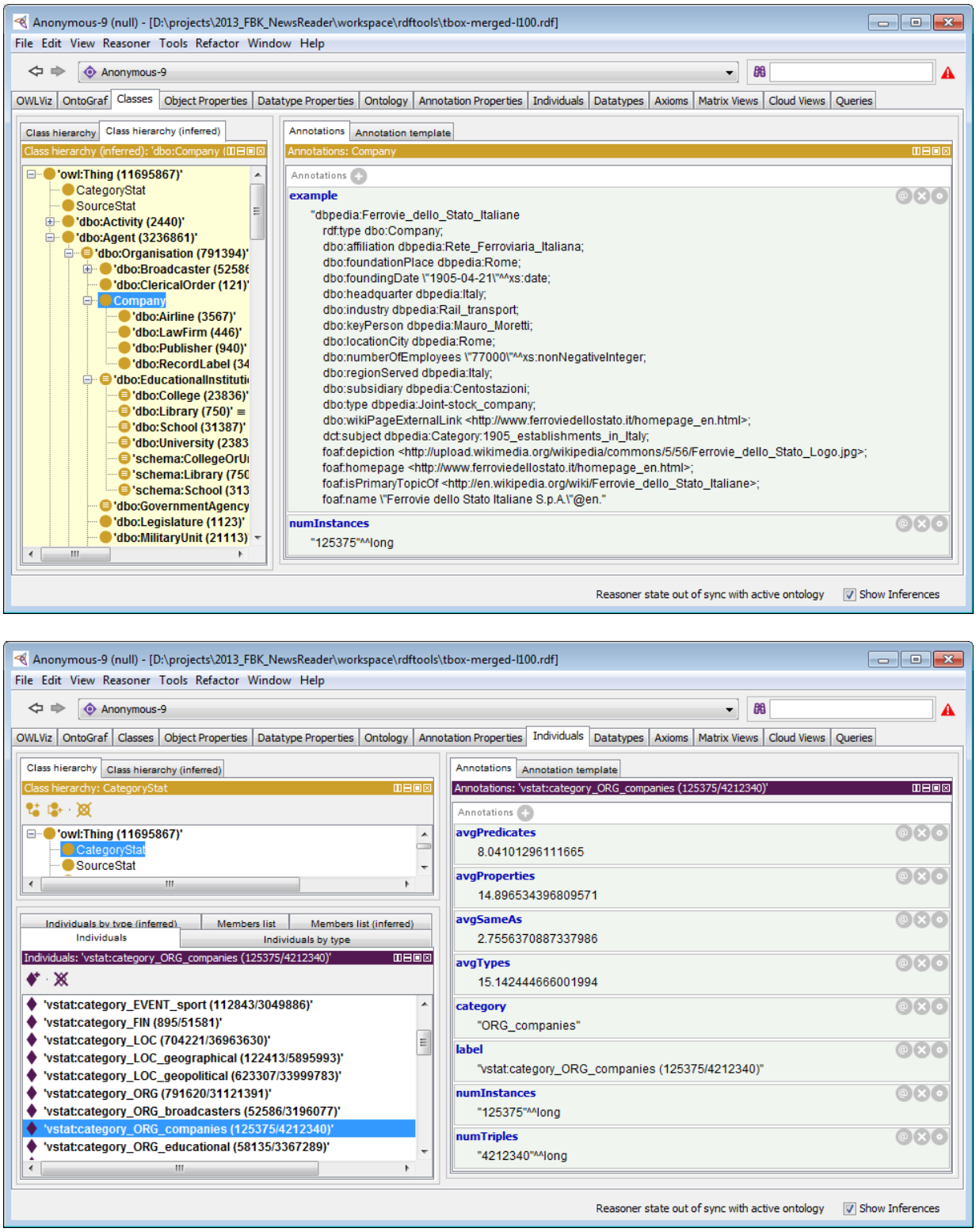

Figure 18: Examples of browsing the statistics ontology in Protégé.

6 Conclusions and Future Work

In this deliverable we documented the first implemented version of the NewsReader KnowledgeStore, a framework enabling to jointly store, manage, retrieve, and semantically query, both unstructured and structured content. The KnowledgeStore plays a central role in the NewsReader project: it stores all contents that have to be processed and produced in order to extract knowledge from news, and it provides a shared data space through which the various NewsReader components (e.g., NLP pipelines, decision support system) cooperate.

We described the changes performed to the KnowledgeStore Data Model, Interfaces, and internal Architecture, with respect to the KnowledgeStore design presented in D6.1. We provided details on the first implementation cycle of the KnowledgeStore, introducing the KnowledgeStore populators, the tools supporting the filling of the KnowledgeStore with documents annotated according to NAF, and structured resources available in RDF format. Furthermore, we described the first collection from LOD sources of background knowledge to be injected into the KnowledgeStore, detailing the selection process and the processing pipeline to extract, combine and augment relevant data to produce a coherent dataset.

This first version of the NewsReader KnowledgeStore provides the core infrastructure functionalities on which the next KnowledgeStore releases will build upon: in particular, the next release (KnowledgeStore version 2, M24) will include reasoning services on the semantic content stored within the KnowledgeStore. As among the goals of NewsReader is to produce scalable results and tools, we are looking forward to deploy and evaluate the KnowledgeStore capabilities and performances in terms on scalability (KnowledgeStore version 3, M33) in the communication and compute infrastructure made available within the context of the Enlighten Your Research 4 (The Big Data Challenge) initiative: from a functional perspective, we plan to assess the KnowledgeStore capability to (i) store an overwhelming daily stream of economical and financial contents (news articles and data), (ii) support a complex NLP pipeline in extracting knowledge from those contents, and (iii) provide suitable online and offline query capabilities for use in a decision support tool for professional decision-makers; from a performance point of view, we plan to evaluate the KnowledgeStore in terms of scalability with respect to data size, query load, and tolerance to nodes and network failures.

References

[Auer *et al.*, 2007] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. In *Proc. of 6th Int. Semantic Web Conference (ISWC'07) and 2nd Asian Semantic Web Conference (ASWC'07), Busan, Korea*, pages 722–735, Berlin, Heidelberg, 2007. Springer-Verlag. http://dl.acm.org/citation.cfm?id=1785162.1785216.

[Beckett, 2004] Dave Beckett. RDF/XML syntax specification (revised). Recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[Bollacker *et al.*, 2008] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: A collaboratively created graph database for structuring human knowledge. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'08)*, pages 1247–1250, New York, NY, USA, 2008. ACM. http://doi.acm.org/10.1145/1376616.1376746.

[Bozzato and Serafini, 2013] Loris Bozzato and Luciano Serafini. Materialization calculus for contexts in the semantic web. In *Proc. of 26th Int. Workshop on Description Logics, Ulm, Germany, July 23 - 26, 2013*, volume 1014 of *CEUR Workshop Proceedings*, pages 552–572. CEUR-WS.org, 2013. http://ceur-ws.org/Vol-1014/paper_51.pdf.

[Bryl *et al.*, 2010] Volha Bryl, Claudio Giuliano, Luciano Serafini, and Kateryna Tymoshenko. Supporting natural language processing with background knowledge: Coreference resolution case. In *Proc. of 9th Int. Semantic Web Conference (ISWC'10)*, volume 6496 of *LNCS*, pages 80–95. Springer, 2010. http://dx.doi.org/10.1007/978-3-642-17746-0_6.

[Buitelaar and Cimiano, 2008] Paul Buitelaar and Philipp Cimiano, editors. *Ontology Learning and Population: Bridging the Gap between Text and Knowledge*, volume 167 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, 2008. http://www.iospress.nl/loadtop/load.php?isbn=9781586038182.

[Carroll *et al.*, 2005] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *Proc. of the 14th Int. Conference on World Wide Web (WWW'05)*, pages 613–622, New York, NY, USA, 2005. ACM. http://doi.acm.org/10.1145/1060745.1060835.

[Cattoni *et al.*, 2012] Roldano Cattoni, Francesco Corcoglioniti, Christian Girardi, Bernardo Magnini, Luciano Serafini, and Roberto Zanoli. The KnowledgeStore: an entity-based storage system. In *Proc. of the 8th Int. Conf. on Language Resources and Evaluation (LREC'12), Instanbul, Turkey*. European Language Resources Association (ELRA), May 2012. http://www.lrec-conf.org/proceedings/lrec2012/summaries/845.html.

[Corcoglioniti *et al.*, 2013] Francesco Corcoglioniti, Marco Rospocher, Roldano Cattoni, Bernardo Magnini, and Luciano Serafini. Interlinking unstructured and structured knowledge in an integrated framework. In *Proc. of 7th IEEE International Conference on Semantic Computing (ICSC), Irvine, CA, USA*, 2013. (to appear).

[De Bruijn and Heymans, 2007] Jos De Bruijn and Stijn Heymans. Logical foundations of (e)RDF(S): complexity and reasoning. In *Proc. of 6th Int. Semantic Web Conference (ISWC'07) and 2nd Asian Semantic Web Conference (ASWC'07), Busan, Korea*, pages 86–99, Berlin, Heidelberg, 2007. Springer-Verlag. `http://dl.acm.org/citation.cfm?id=1785162.1785170`.

[Feigenbaum *et al.*, 2013] Lee Feigenbaum, Gregory Todd Williams, Kendall Grant Clark, and Elias Torres. SPARQL 1.1 protocol. Recommendation, W3C, March 2013. `http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/`.

[Ferrucci *et al.*, 2010] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A. Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John Prager, Nico Schlaefer, and Chris Welty. Building Watson: An overview of the DeepQA Project. *AI Magazine*, 31(3), 2010. `http://www.aaai.org.proxy.lib.sfu.ca/ojs/index.php/aimagazine/article/view/2303`.

[Gantz and Reinsel, 2011] John Gantz and David Reinsel. Extracting value from chaos. Technical report, IDC Iview, June 2011. `http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf`.

[Gilbert and Lynch, 2002] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. `http://doi.acm.org/10.1145/564585.564601`.

[Hoffart *et al.*, 2011] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, Edwin Lewis-Kelham, Gerard de Melo, and Gerhard Weikum. YAGO2: exploring and querying world knowledge in time, space, context, and many languages. In *Proc. of 20th Int. Conf. companion on World Wide Web (WWW'11), Hyderabad, India*, pages 229–232, New York, NY, USA, 2011. ACM. `http://doi.acm.org/10.1145/1963192.1963296`.

[Hoffart *et al.*, 2013] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence*, 194:28–61, 2013. `http://dx.doi.org/10.1016/j.artint.2012.06.001`.

[Motik *et al.*, 2009] Boris Motik, Bijan Parsia, and Peter F. Patel-Schneider. OWL 2 Web Ontology Language structural specification and functional-style syntax. Recommendation, W3C, October 2009. `http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/`.

[Palmero Aprosio *et al.*, 2013] Alessio Palmero Aprosio, Claudio Giuliano, and Alberto Lavelli. Towards an automatic creation of localized versions of dbpedia. In *Proc of 12th Int. Semantic Web Conference (ISWC'13)*, volume 8218 of *Lecture Notes in Computer Science*, pages 494–509. Springer Berlin Heidelberg, 2013. http://dx.doi.org/10.1007/978-3-642-41335-3_31.

[Patel-Schneider and Franconi, 2012] Peter F. Patel-Schneider and Enrico Franconi. Ontology constraints in incomplete and complete data. In *Proc. of the 11th Int. Semantic Web Conference (ISWC'12), Boston, MA*, pages 444–459, Berlin, Heidelberg, 2012. Springer-Verlag. http://dx.doi.org/10.1007/978-3-642-35176-1_28.

[Stadler *et al.*, 2012] Claus Stadler, Jens Lehmann, Konrad Höffner, and Sören Auer. LinkedGeoData: A core for a web of spatial open data. *Semantic Web Journal*, 3:333–354, 2012. http://www.semantic-web-journal.net/sites/default/files/swj173_2.pdf.

[Tao *et al.*, 2010] Jiao Tao, Evren Sirin, Jie Bao, and Deborah L. McGuinness. Integrity constraints in OWL. In *Proc. of 24th Conf. on Artificial Intelligence (AAAI'10), Atlanta, Georgia, USA*. AAAI Press, July 2010. http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1931.

[van Hage *et al.*, 2011] Willem Robert van Hage, Véronique Malaisé, Roxane Segers, Laura Hollink, and Guus Schreiber. Design and use of the Simple Event Model (SEM). *J. Web Sem.*, 9(2):128–136, 2011. http://dx.doi.org/10.1016/j.websem.2011.03.003.