Knowledge store version 2

Deliverable D6.2.2

Version FINAL

Authors: Francesco Corcoglioniti[1], Marco Rospocher[1], Roldano Cattoni[1], Marco Amadori[1], Bernardo Magnini[1], Mohammed Qwaider[1], Michele Mostarda[1], Alessio Palmero Aprosio[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 D6.2.2
Status & Version	FINAL
Contractual Date of Delivery	January 2015
Actual Date of Delivery	January 30, 2015
Type	Prototype
Security (distribution level)	Public
Number of Pages	87
WP Contributing to the Deliverable	WP6
WP Responsible	FBK
EC Project Officer	Susan Fraser

Authors: Francesco Corcoglioniti[1], Marco Rospocher[1], Roldano Cattoni[1], Marco Amadori[1], Bernardo Magnini[1], Mohammed Qwaider[1], Michele Mostarda[1], Alessio Palmero Aprosio[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 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	18 December 2013	Starting new draft from D6.2.1	Francesco Corcoglioniti, Marco Rospocher	
0.2	20 July 2014	Added user interface	Francesco Corcoglioniti, Alessio Palmero Aprosio	3.3
0.3	15 August 2014	Added backends	Alessio Palmero Aprosio	4.1.5
0.3.1	30 November 2014	Updated NAF populator	Roldano Cattoni	5
0.4	15 December 2014	Added RDFpro	Francesco Corcoglioniti, Marco Rospocher, Alessio Palmero Aprosio	6
0.5	20 December 2014	Added ESO reasoner	Alessio Palmero Aprosio	7
1.0	10 January 2015	Revision of whole document	Alessio Palmero Aprosio	
1.0.1	14 January 2015	Updated background knowledge	Francesco Corcoglioniti	5.3
1.1	20 January 2015	Added use cases	Francesco Corcoglioniti, Marco Rospocher, Alessio Palmero Aprosio	5.4
1.1	25 January 2015	Review minor editorial changes	Antske Fokkens	All
1.1	30 January 2015	Check by coordinator	VUA	-

Executive Summary

This deliverable documents the second 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 this framework. We also highlight the key role of the KnowledgeStore in achieving challenging goals of the NewsReader project.

We provide a detailed description of 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 outline the internal component organization of the KnowledgeStore (the KnowledgeStore *architecture*), discussing the technological and implementation choices we made. Then, we present the KnowledgeStore *populators* which are the tools that process annotated news documents and structured resources to fill the KnowledgeStore with content: in particular, in this 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], ACM SAC 2015 [Corcoglioniti *et al.*, 2015], ISWC 2014 Posters and Demonstrations Track [Rospocher *et al.*, 2014a], Posters and Demos of the 19th International Conference on Knowledge Engineering and Knowledge Management (EKAW2014) [Rospocher *et al.*, 2014b], and ISWC Developers Workshop [Corcoglioniti *et al.*, 2014a].

Contents

List of Figures

1 Introduction

This prototype deliverable presents the implementation of the second 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 D6.2.1. This revision updates the KnowledgeStore design in light of the latest outcomes of some activities: the implementation of the user interface (Section 3.3); some updates on the population (including three different case studies, showing the scalability of the KnowledgeStore, see Section 5); the description of some new features in the KnowledgeStore architecture (Section 4); the development of a tool (RDF_{PRO}, see Section 6); the new ESO reasoner (Section 7).

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 RDF_{PRO} tool, available at `http://fracor.bitbucket.org/rdfpro/`, and its ESO reasoner processor (`https://bitbucket.org/aprosio/eso-reasoner/`).

The current deliverable will serve as basis for the documentation of all the next development cycles of the KnowledgeStore, and will be updated and integrated in the future Deliverable D6.2.3 to describe the reasoning service built on top of it (M33, Deliverable D6.2.3, final scalable version). Descriptions from Deliverable D6.2.1 have been taken up in this deliverable. We have added a short overview of new additions to the deliverable at the beginning of each section other than this introduction and the conclusion. These indications are meant to point readers familiar with the content of D6.2.1 quickly to relevant parts of this updated version.

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[1] initiative),

[1] `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 concerns 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 from 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, we still lack frameworks enabling the seamless integration and linking of knowledge coming both from structured and unstructured content.

This document describes the implementation 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 by 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 for 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

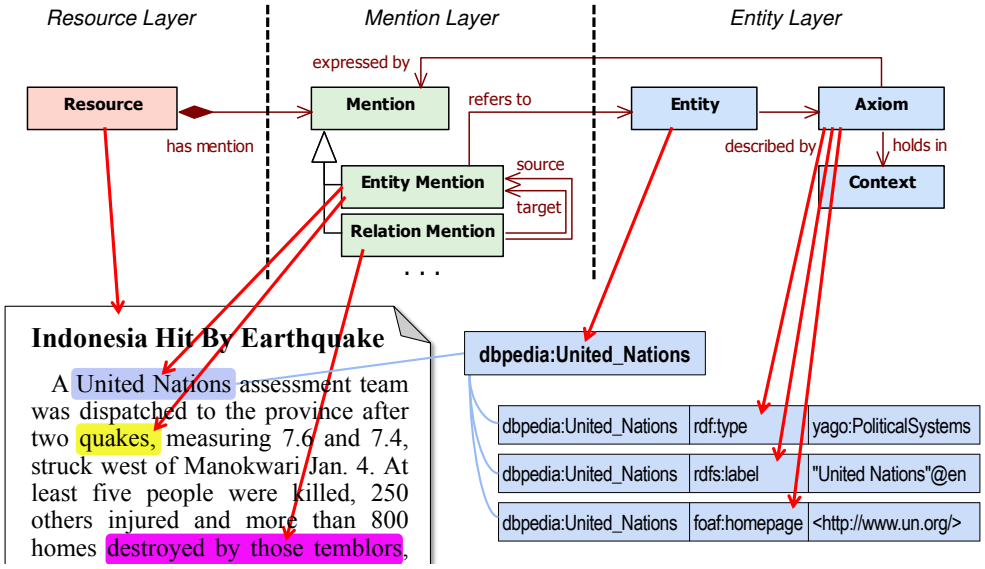


Figure 1: KnowledgeStore Content.

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 of semantically querying 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 and/or 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 or 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's and 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.[2] 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[3] 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, who 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;

[2] http://www.livememories.org/
[3] 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,[4] 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.[5]

First, the KnowledgeStore allows us 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*[6] 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 linguistically annotated news documents[7] 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.

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

[5] 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.

[6] 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.

[7] These documents represent text and linguistic annotations in the NLP Annotation Format ([Fokkens *et al.*, 2014], NAF)

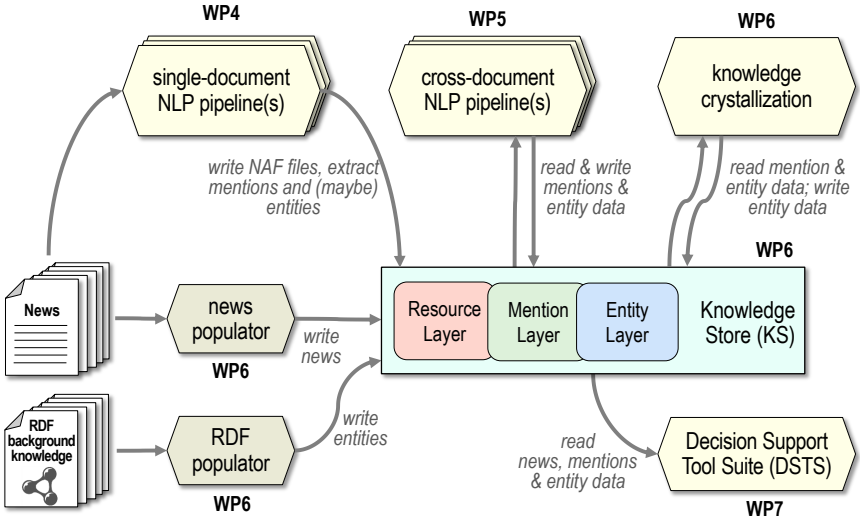


Figure 2: The role of the KnowledgeStore in NewsReader.

- *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 needs to be investigated. Therefore the KnowledgeStore currently does not handle them, although it does offer the basic operations to implement the more appropriate strategy to cope with them to each module.

2 The KnowledgeStore Data Model

Changes wrt the KnowledgeStore Data Model described Deliverable D6.2.1

- Added references to Grounded Annotation Framework and updated pictures describing the Data Model, with the addition of `gaf:` namespace (Figures 3 and 7).

The data model defines what information can be stored in the KnowledgeStore, in accordance with the annotation guidelines of WP3,[9] the event modeling activity of WP5 and the NLP Annotation Format ([Fokkens *et al.*, 2014], NAF) and Grounded Annotation Framework ([Fokkens *et al.*, 2013], GAF) developed as part 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 instance data in RDF [Beckett, 2004], which in turn enables interoperability with Semantic Web applications and technologies.

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

[10] NAF (NLP 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 is described in Deliverables D2.1 and D2.2: System Design.

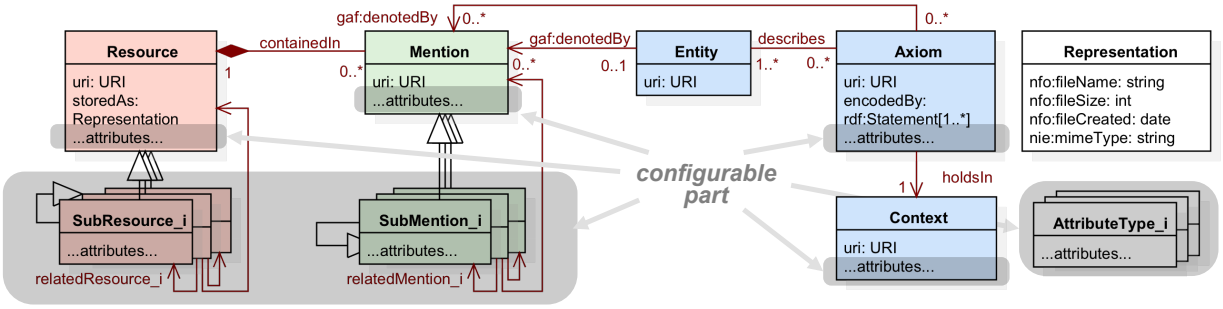


Figure 3: KnowledgeStore data model.

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. Entities can be denoted by one or several mentions, indicated by the `gaf:denotedBy` relation.
- 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 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 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

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

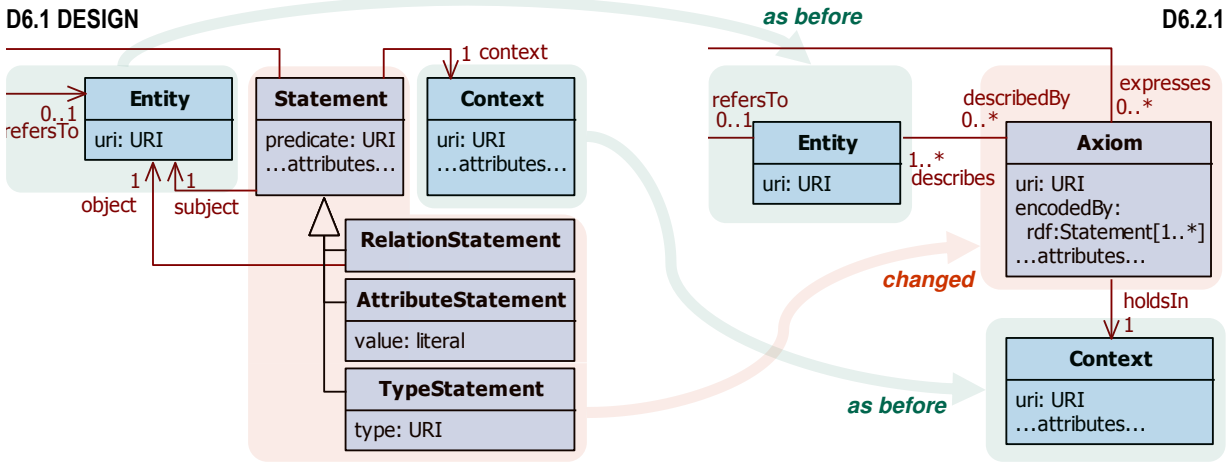


Figure 4: From RDF statements to axioms.

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 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 us to put multiple axioms in the same module in case they share the same context and

```
@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.

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 (see Deliverable D3.3.2: Annotated Data) as well as the latest NAF specification (se Deliverable D2.2: 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 article, 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. Some more details are provided below:

- News articles 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.

Mention layer The position of a mention in a news article 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 gaf:denotedBy from entity to mention). 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:

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

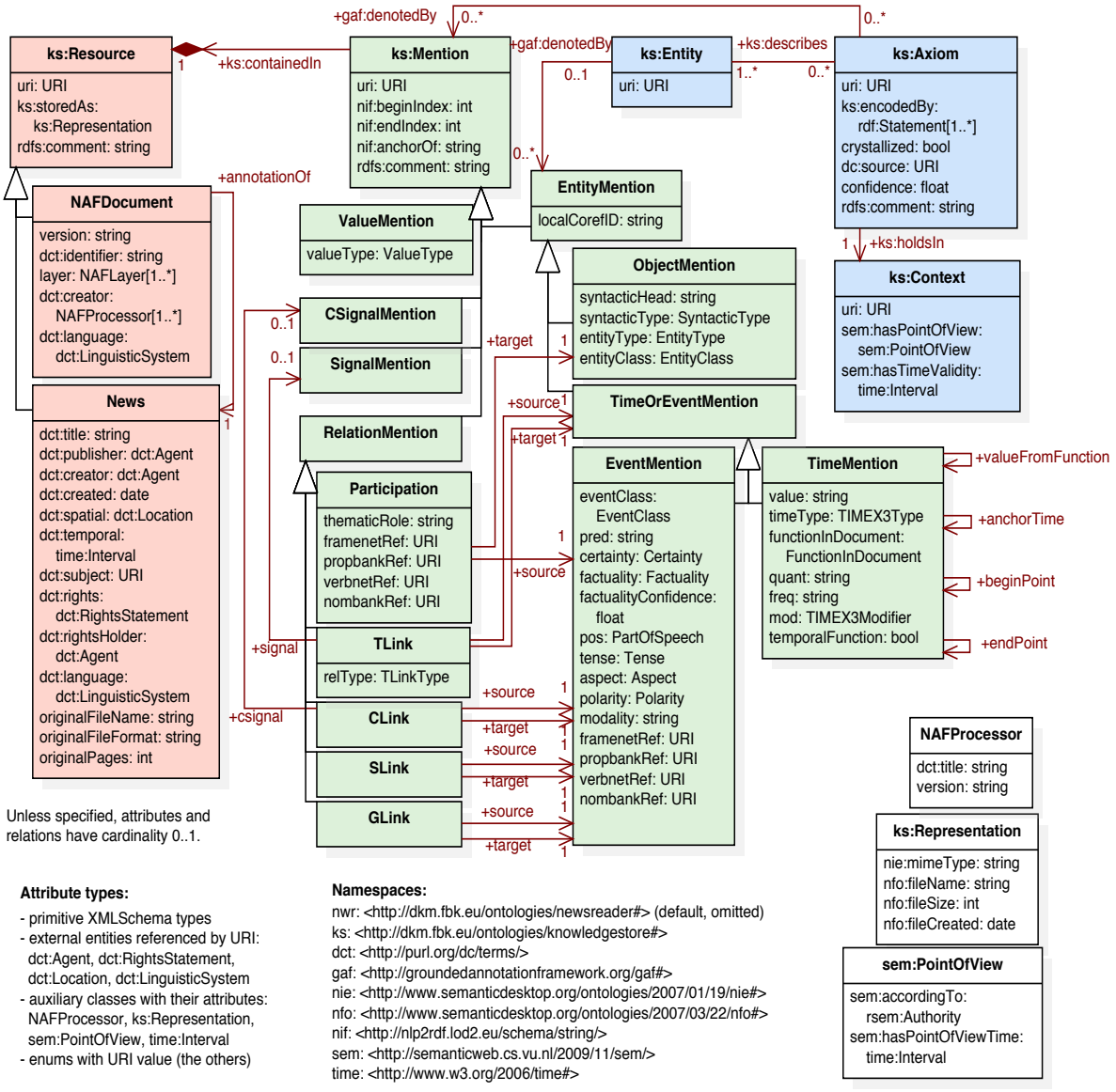


Figure 7: NewsReader data model.

  - 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).
  - 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, geo-political 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 D6.2.1

- introduced a new section on the KnowledgeStore User Interface.

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 this section.

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 operations 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 operations 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 of selecting 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.

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 is therefore used

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

in the current implementation 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 compatible 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

[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.

[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

or may cause the API request to fail.

Security Access to the KnowledgeStore API must be restricted only to authorized clients, since it allows for 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 analyze 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 analyzed[23] to find commonalities, in order to remove duplicates or operations subsumed by other ones. By adopting a generalization 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

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.

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

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 detail, 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 the 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 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.

[25] Please refer to the online documentation for the full condition syntax.

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 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 9 illustrates the use of the KnowledgeStore Java client within an application for retrieving all the mentions of type nwr:entity_type_per.

[26] With respect to the 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.

```
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.

3.2.2 SPARQL Endpoint

The SPARQL endpoint allows users 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

[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.

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 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.

3.3 The KnowledgeStore User Interface

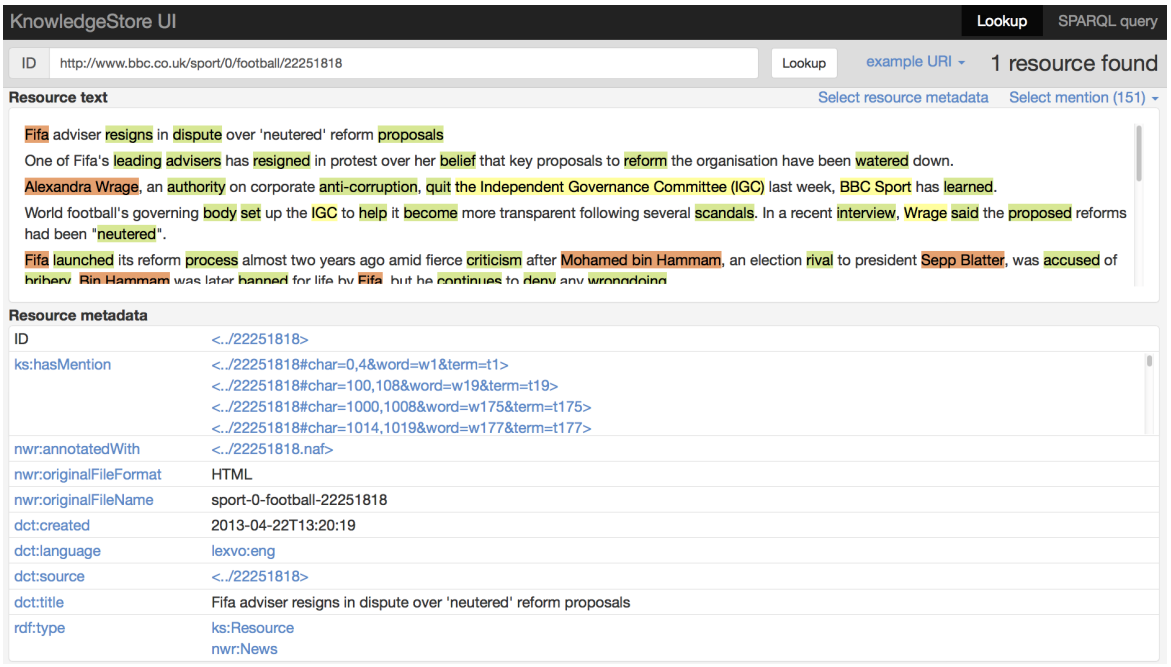
While the KnowledgeStore can be programmatically accessed through its API, human users can easily inspect and navigate the KnowledgeStore content through the KnowledgeStore *User Interface* (UI).[29] The KnowledgeStore UI is a web-based application that offers two core operations:

- the *SPARQL query* operation, with which arbitrary SPARQL queries can be run against the KnowledgeStore SPARQL endpoint, obtaining the results directly in the browser or as a downloadable file (in various file formats, including the recently standardized JSON-LD). Figure 11c shows an excerpt of the result set obtained by running a query in the SPARQL tab of the KnowledgeStore UI;

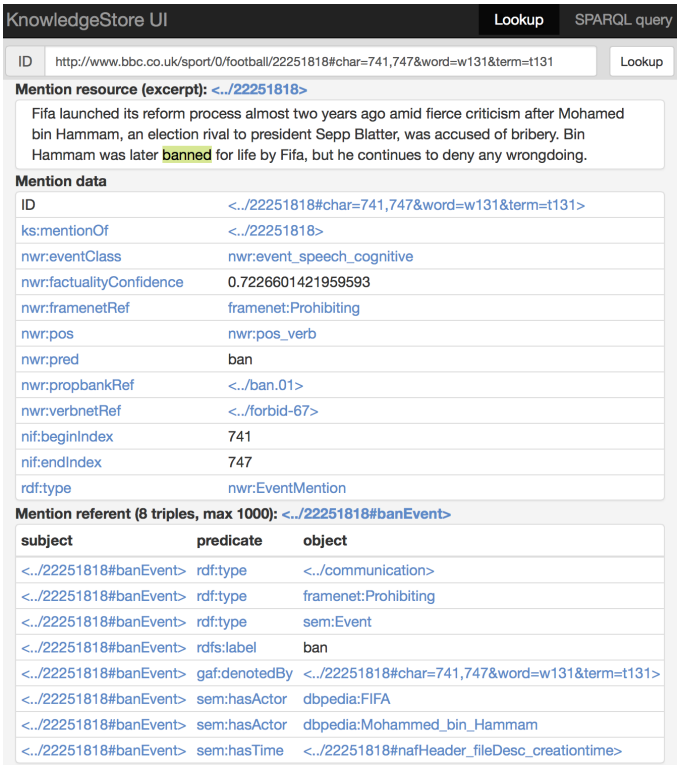
- the *lookup* operation, which given the URI of an object (i.e., resource, mention, entity), retrieves all the KnowledgeStore content about that object. Figure 11a and Figure 11b show the output obtained by running a lookup operation for a resource and for a mention.

These two operations are seamlessly integrated in the UI, to offer a smooth browsing experience to the users. For instance, it is possible to directly invoke the lookup operation on any entity returned in the result set of a SPARQL query. Similarly, when performing the lookup operation on a resource, all mentions occurring in the resource are highlighted (see the 'Resource text" box in Figure 11a) with a different color for the various mention types (e.g., person, organization, location, event), and by clicking on any of them the user

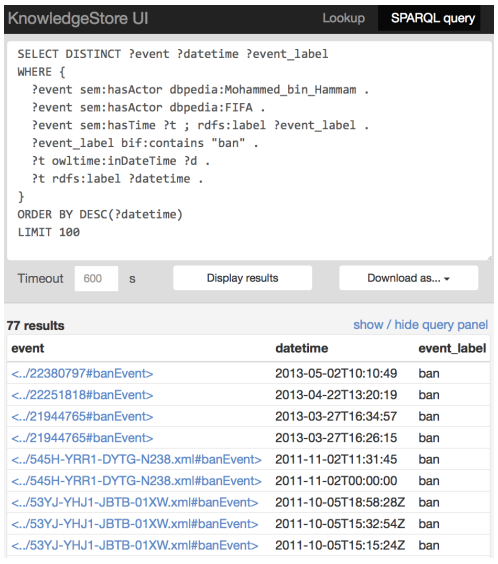
[29] The features of the KnowledgeStore UI are comprehensively showcased in a demo video `http://youtu.be/if1PRwSll5c`.



(a) Resource Lookup



(b) Mention Lookup



(c) Running a SPARQL query

Figure 11: The KnowledgeStore UI

can access all the details for that mention (see Figure 11b). The lookup operation on a mention properly remarks the three distinct representation layers of the KnowledgeStore (note the three boxes—*Mention resource*, *Mention Data*, *Mention Referent*—in Figure 11b corresponding to the three representation layers of the KnowledgeStore), and the role of mentions as a bridge between unstructured and structured content.

4 The KnowledgeStore Architecture and Implementation

Changes wrt the KnowledgeStore Architecture described Deliverable D6.2.1

- added description of the `MultiFileStore`, to obtain better performances by HDFS (Section 4.1.2);
- added Section 4.1.5 on alternative backends for KnowledgeStore installation on single-machine environment;
- updated description of software development (Section 4.2.2).

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.[30]

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 12 shows the overall KnowledgeStore architecture, highlighting its client-server nature.

Client side The client side (upper part of Figure 12) 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[31] for accessing the SPARQL endpoint, thanks to its standard-based nature. From a functional point of view, client application may carry out different tasks:

- *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;

[30] `http://newsreader.fbk.eu/knowledgestore`
[31] See `http://www.w3.org/wiki/SparqlImplementations`.

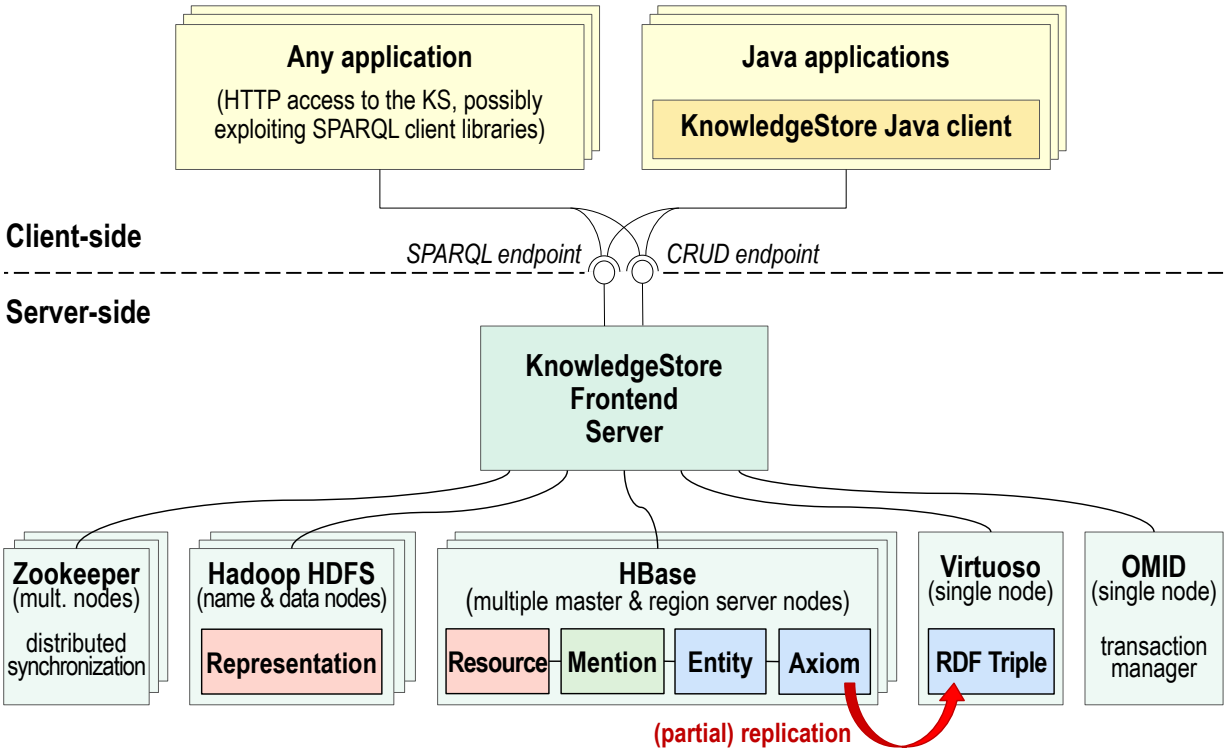


Figure 12: KnowledgeStore architecture.

- *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 12) 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[32] is used in combination with HBase to enforce the

[32] `https://github.com/yahoo/omid`

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 12 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 sections, we present the main server-side software components of the KnowledgeStore architecture, namely HBase & Hadoop (Section 4.1.1), Virtuoso (Section 4.1.3) and the KnowledgeStore Frontend Server (Section 4.1.4).

4.1.1 HBase & Hadoop

Hadoop[33] and HBase[34] 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. The encoding is based on schemas compliant with the *Apache Avro*[35] 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

[33] `http://hadoop.apache.org`
[34] `http://hbase.apache.org`
[35] `http://avro.apache.org/`

of the KnowledgeStore API may however affect multiple rows in different tables for each modified object, as happens, 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,[36] which provides a full transaction manager over HBase. OMID exploits the versioning capabilities of HBase to realize a Multiversion Concurrency Control (MVCC) mechanism[37] 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 13 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 users to lookup the description of an entity in a single, more efficient operation. Figure 13 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.[38]

4.1.2 The `MultiFileStore`

The Hadoop Distributed File System (HDFS) is a reliable cloud storage platform, but is designed especially for very large files. As there is a single NameNode that stores metadata in its main memory, performances drop as the number of files increases (see [Chandrasekar *et al.*, 2013]). As a consequence, storing a big number of small files is the worst scenario in a HDFS environment, and for the KnowledgeStore this is an issue to take into consideration.

Proposed solutions deal with this HDFS issue by merging small files into a minor number of larger files [Dong *et al.*, 2010]. In particular, for the KnowledgeStore, a new module called `MultiFileStore` has been developed. When a file is added to the KnowledgeStore, it is

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

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

[38] 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.

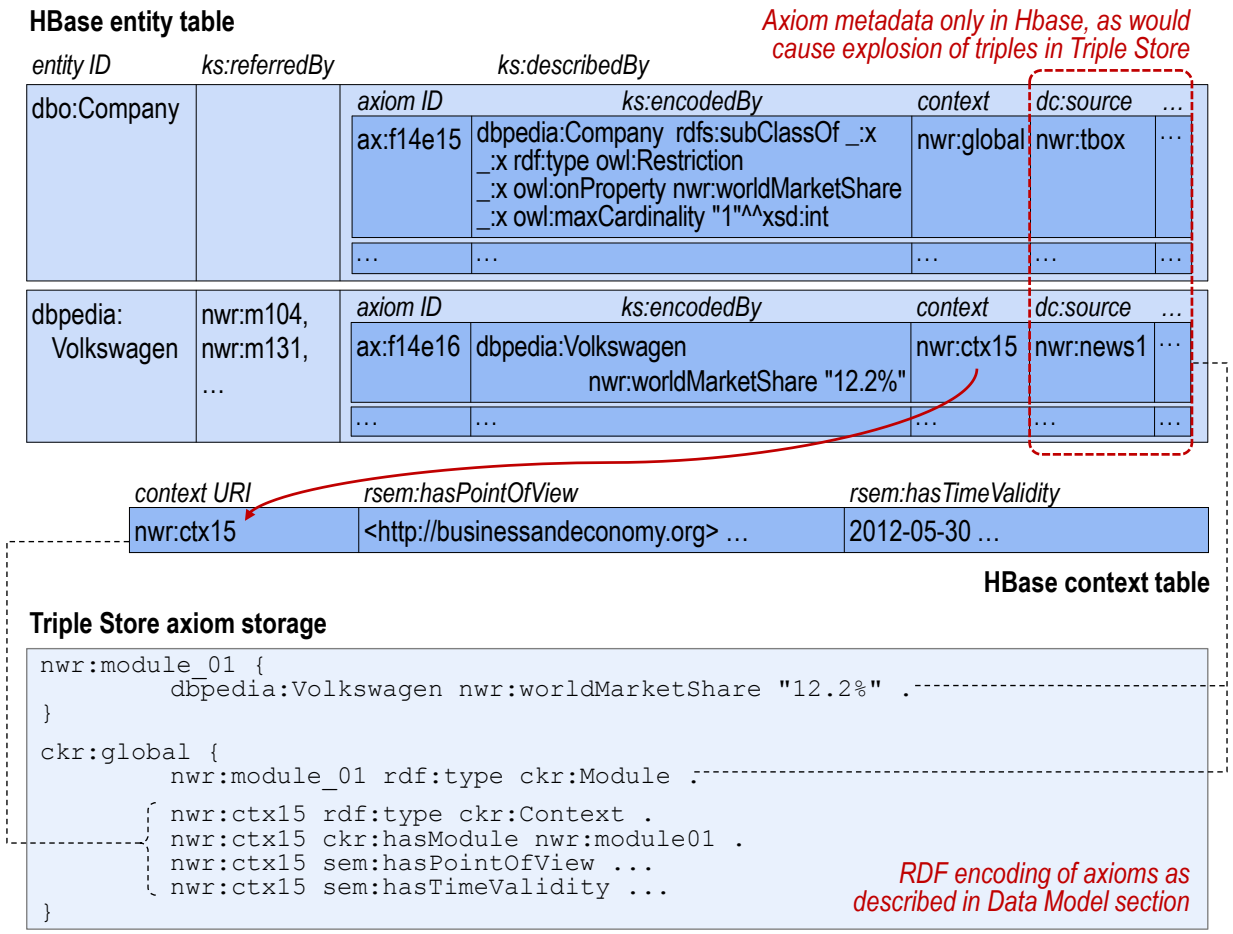


Figure 13: Axiom representation in HBase and in the Virtuoso Triple Store.

first collected in a particular working folder. When the number of files contained in the folder is greater than k, a background process starts and merge these files into a big zip file and save it in the production folder. Then the list of small files is saved into a Apache Lucene[39] index. This index is then used to retrieve the file when needed. Finally the small files are removed from the working folder.

The parameter k (number of small files for each phisical file) can be set in the KnowledgeStore configuration and, depending on that, the performances of the HDFS may change (see Table 1).

4.1.3 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 using the RDF representation described in Section 2.1, i.e., as sets of ⟨subject, predicate, object⟩

[39] `https://lucene.apache.org/`

Table 1: Performance of the `MultiFileStore` (varying the parameter k) on a small dataset that includes 330 files. For reference, without the `MultiFileStore`, totalAverageTime is 9.8 s (individualAverageTime 30 ms/file).

numSmallFile	totalAverageTime	individualAverageTime
10	13.3 s	40 ms/file
50	28.0 s	85 ms/file
100	45.7 s	138 ms/file

RDF triples within named graphs (the *modules*) that are connected to context definitions in a specific ckr:global graph; as previously anticipated and shown in Figure 13, axiom metadata is not indexed and cannot thus be directly queried using SPARQL. The Open Source Edition of the Virtuoso triple store,[40] version 7.0.0, has been chosen, motivated by its excellent performances in recent (April 2013) benchmarks[41] 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,[42] using the Virtuoso Sesame driver.[43] The Sesame API enables a uniform access to triple stores from Java applications, thus making it easier to replace Virtuoso with a different triple store should the need arise within or beyond NewsReader (e.g., for scaling up but 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 turned out 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 specifically, 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[44]). 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.[45]

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

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

[42] `http://www.openrdf.org/`

[43] 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.

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

[45] 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

```
        ?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 14: Examples of inference rules

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.[46] 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. Customized 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 14 shows two examples of customized inference rules: rule in Figure 14a 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 14b 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).

4.1.4 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[47] standardized by W3C. The CRUD endpoint is instead implemented as an HTTP ReST service using JSON for Linked Data (JSON-LD)[48] as the data format. JSON-LD is a

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.

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

[47] `http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/`

[48] `http://json-ld.org/`

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.

4.1.5 Alternative backends

In the development of the KnowledgeStore, the main issue to take into account is the need of good performances on a big number of files (corresponding to a huge number of mentions). For this reason, the default configuration uses HBase to store mentions and Hadoop to store resources (see Section 4.1.1). To favour the test of the KnowledgeStore on a single machine by a end-user, some new modules have been developed:

- The Hadoop file store also accepts a local folder as parameter: the KnowledgeStore will read/write/delete files in that folder, transparently. It also works with the `MultiFileStore` (see Section 4.1.2).
- In place of HBase, a MySQL[49] database or an Apache Lucene[50] index can be used. The first solution needs MySQL server to be installed on the machine, the second one is self-contained in the KnowledgeStore package and does not need any external software. For small datasets (less than 1,000 documents) it is also available a module called `MemoryDataStore` that keeps the Data Store in memory and optionally loads/saves it to disk on KnowledgeStore launch/shutdown.

Section 4.2.1 describes the inclusion of these new modules in the KnowledgeStore development in more detail.

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,

[49] `http://www.mysql.com/`
[50] `https://lucene.apache.org/`

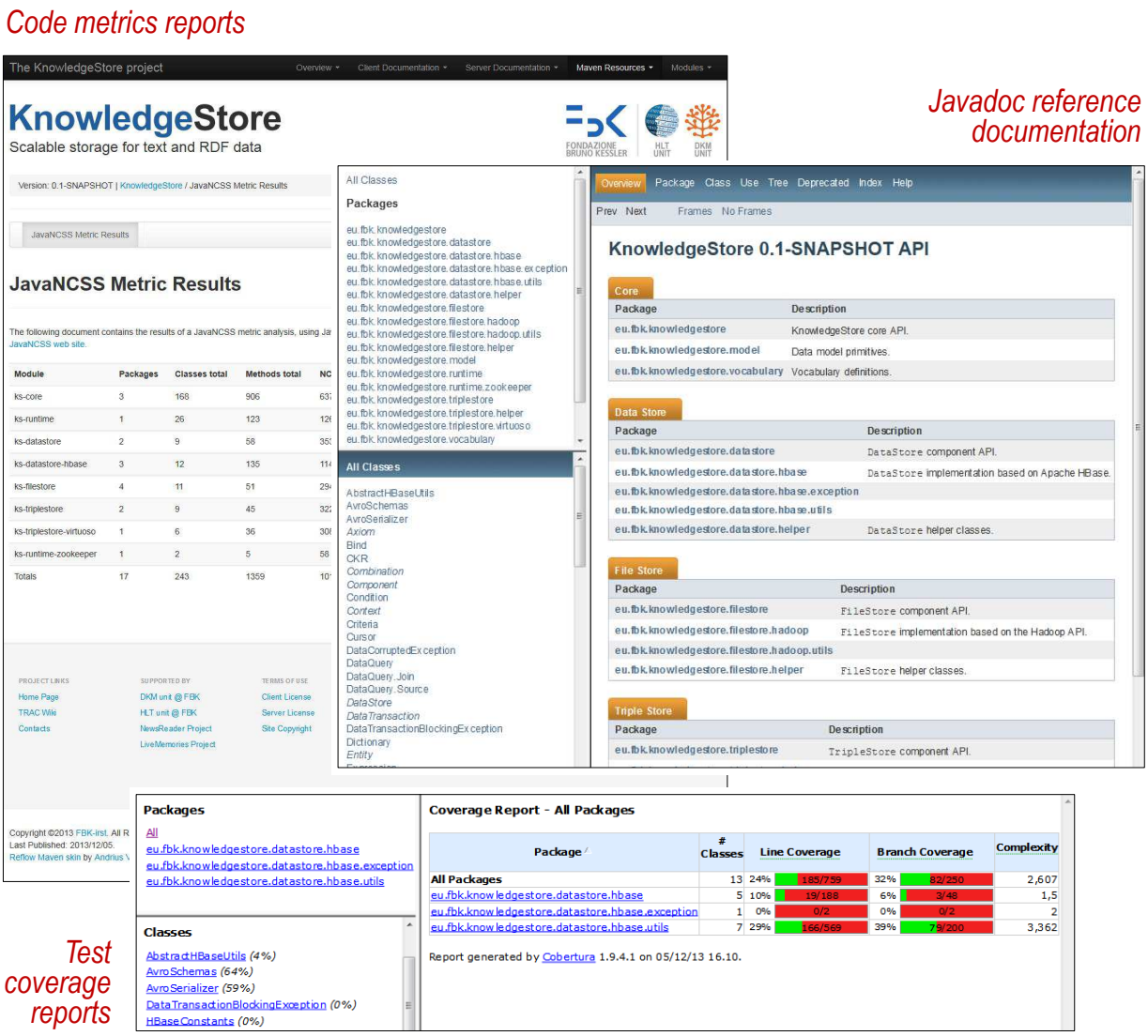


Figure 15: Examples of generated reports on the KnowledgeStore web site.

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 outlined 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 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[51] build system and model have been used to manage the overall source code organization and all the phases of the build lifecycle (compiling, testing, release, ...), in combination with the Eclipse[52] 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[53] 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 15. A Maven *multi-module* project organization has been adopted, with code organized in modules according to a functional criterion, as shown in Figure 16. 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.

[51] `http://maven.apache.org/`
[52] `http://www.eclipse.org/`
[53] `http://newsreader.fbk.eu/knowledgestore`

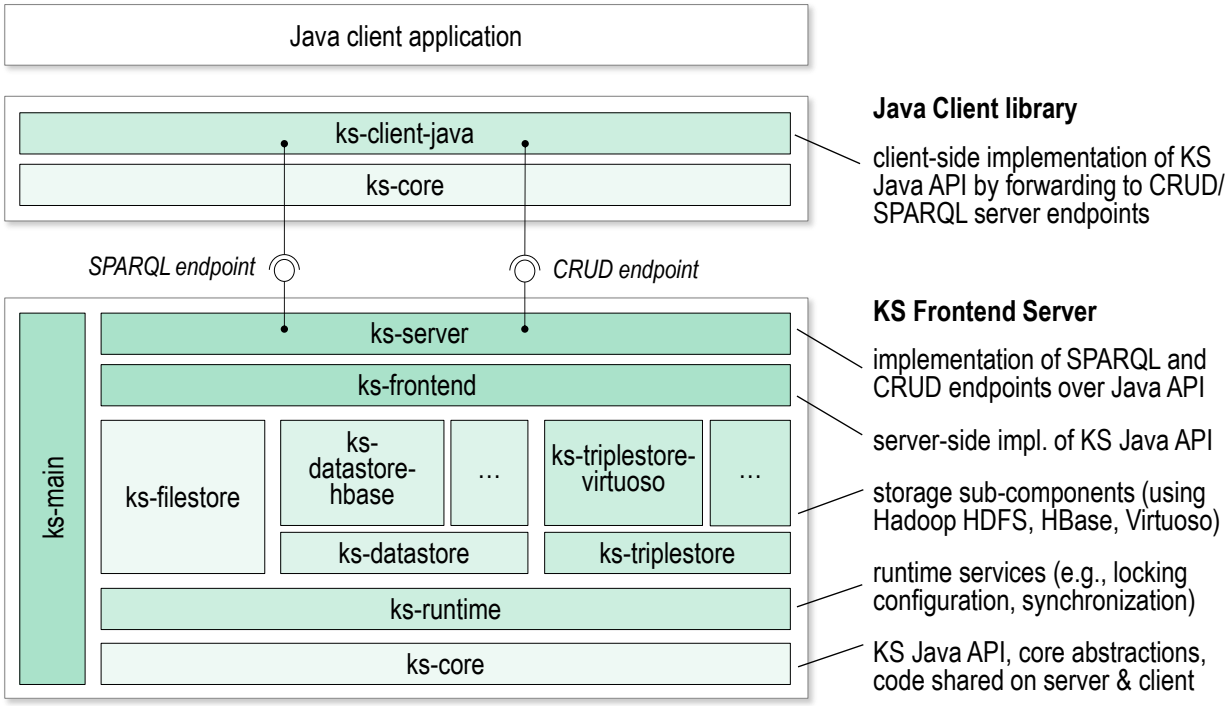


Figure 16: Modular code organization.

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. Three more modules (MySQL, Lucene, and Memory) make local installation and test easier.

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[54] virtual machine. As an alternative the MySQL, Lucene, and Memory data stores can be used without needing the 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.

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

5 The KnowledgeStore Population

Changes wrt the KnowledgeStore Population described Deliverable D6.2.1

- multi-threading implementation of the NAF-populator to improve performance (Section 5.1.1);
- added KnowledgeStore use cases (Section 5.4) and discussion (Section 5.5);
- generation of multiple background knowledge datasets tailored to different usage scenarios: EN-only vs multi-language (EN, ES, IT, NL), with variants enriched with data from other localizations of DBpedia for a total of 18 languages considered (Section 5.3.1);
- update to DBpedia 2014 with more data filtered out based on first year usage experience, resulting in the generation of smaller, more manageable background knowledge datasets (Sections 5.3.1 and 5.3.3);
- revision of the pipeline for the generation of background knowledge datasets, now based on the RDF$_{\text{PRO}}$ tool (factored out from first year pipeline) plus simple script and configuration data, all released on the KnowledgeStore web site to allow for third party reuse/customization (Section 5.3.2).

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 NLP Annotation Format ([Fokkens *et al.*, 2014], 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 NLP 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.

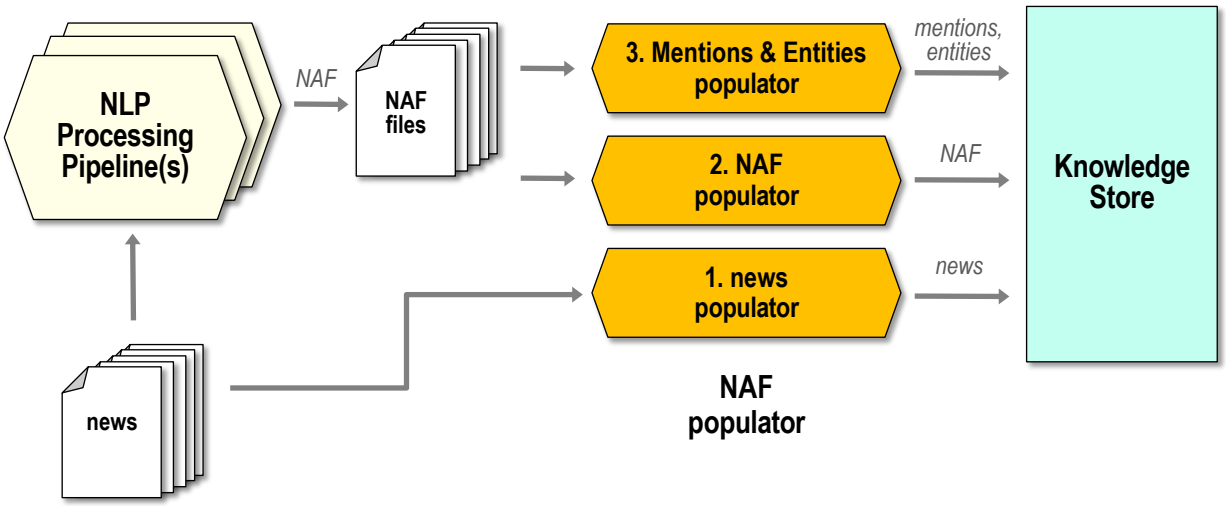


Figure 17: NAF population.

As shown in figure 17, 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:

```
<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.[55] 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 quantity 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

[55] `http://tools.ietf.org/html/rfc5147`

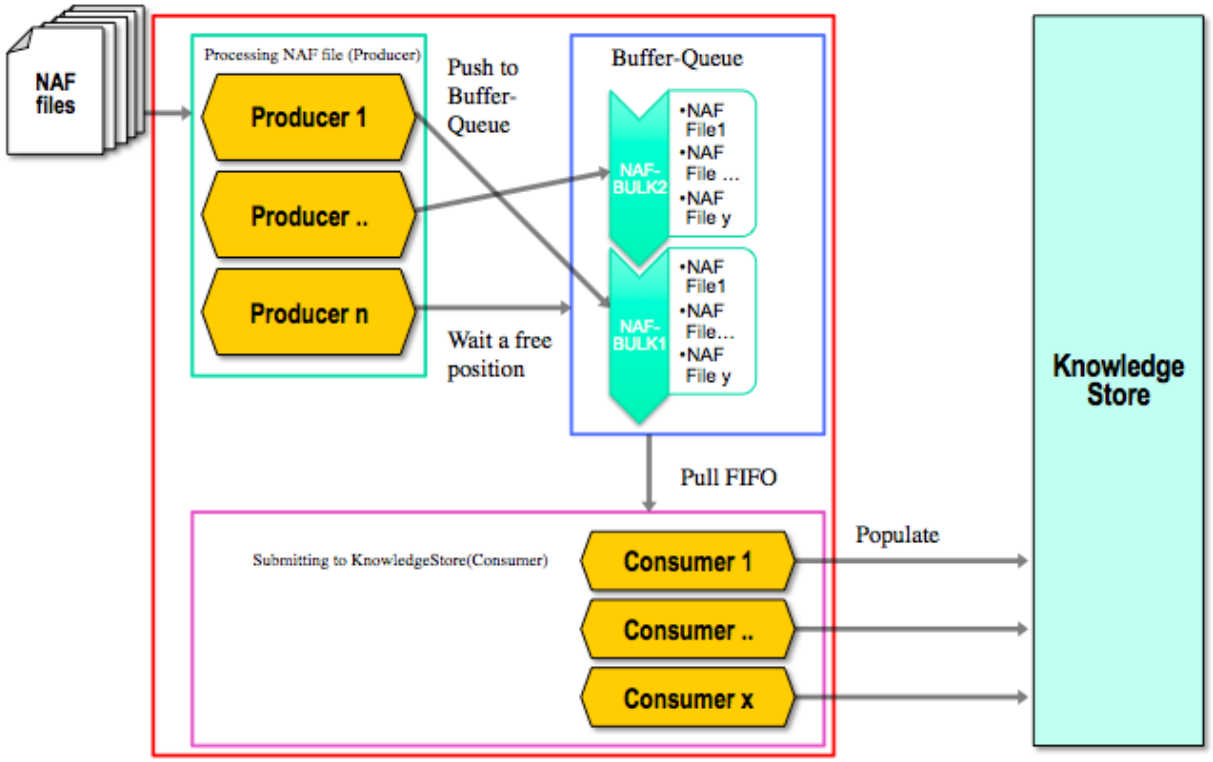


Figure 18: NAF Multi-Threading populator.

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 time 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.

5.1.1 Multi-Threading

While dealing with big data, many issues are considered crucial, such as: NAF processing time, populator-KnowledgeStore interaction time and memory size.

In this respect, we have implemented a homogeneous robust asynchrony multi-threading application based on the Producer-Consumer paradigm,[56] where its three main components (Buffer-Queue, Producers and Consumers) can be customized.

Figure 18 shows the internal architecture of the NAF-populator whose main components are:

[56] http://en.wikipedia.org/wiki/Producer-consumer_problem

- Buffer-Queue: this is the queue which holds the information extracted from NAF files that are ready to be sent to the KnowledgeStore to be stored; each element of the queue contains a bulk of NAFs and the information extracted from them (namely a set of resources and a set of mentions). The number of elements of the Buffer-Queue can be customized using the option "-q INT" (default to 1). Also the size of the bulk can be customized using the option "-b INT" (default to 1).
- Producers: they are the modules that process the NAF files and extract the information according to the KnowledgeStore data model (see section 2). If the Buffer-Queue has at least one empty element, a free producer takes care of processing a new bulk of NAFs and push the results into Buffer-Queue, otherwise it sleeps till an empty element is available.
- Consumers: they are the modules that take the information extracted from the NAFs by the producers and send it to the the KnowledgeStore to be saved. Consumers directly interact with the KnowledgeStore through its API. Multiple consumers can be instantiated with the option "-ct INT" (default to 1).

Before implementing this architecture, we conducted some timing experiments to better understand the population activities: it has emerged that the interaction with the KnowledgeStore is much more time-consuming than the processing of NAFs. In other words, having multiple producers do not improve the overall performance, because the bottleneck is in the consumer activity. This is the reason why we implemented multiple consumers with the hope that the KnowledgeStore can exploit such parallelization. In any case, extending the NAF-populator with multiple producers is not a difficult task. Concerning the Buffer-Queue size, this parameter, together with the bulk size, are crucial for the application because they highly impact on the memory: using too high size may produce a memory crash.

5.2 RDF populator

The RDF populator (sources and binaries available online[57]) 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:

- 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.

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

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.[58] Contexts URIs in input data are also ignored, as the KnowledgeStore automatically generates them based on the values of contextual dimensions. To be precise, 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,[59] which performs in-memory sort and falls back 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).

[58] 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.

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

5.3 Acquisition of LOD background knowledge

Background knowledge consists of terminological and assertional data that describes entities, events, their relations and structure and supports data-intensive applications and processing tasks like the ones carried out in NewsReader. A prominent source of background knowledge is the Linked Open Data (LOD) cloud: a collection of RDF data about entities in different domains consisting of over 74 billions of triples in ∼1K interlinked datasets.[60] Although this data presents shallow structure and semantics, the wealth of information conveyed, and the fact that data is constantly updated (mainly through community efforts) make this kind of data particularly useful for NewsReader. For this reason, background knowledge from LOD sources is collected as part of WP6 and stored in the KnowledgeStore, where it is integrated with knowledge extracted from texts and made available for consumption to use case applications (e.g., the hackathon ones) and NLP modules.

This section contains an up-to-date description (compared to deliverable D6.2.1) of the activities for the collection of background knowledge carried out in WP6. Section 5.3.1 focuses on the *data selection* process, reporting on the adopted selection criteria and describing the different selections chosen for import in the KnowledgeStore and suited to different usage scenarios. Section 5.3.2 describes the *data integration* process implemented to extract, combine and augment selected data in order to produce the different background knowledge datasets. Section 5.3.3 describes the resulting datasets, reporting on both data statistics and processing times.

5.3.1 Data selection

Importing all LOD data as background knowledge is unfeasible due to its huge size, and also because not all of this data is relevant for NewsReader. A selection of data is thus necessary and can be guided by the set of *selection criteria*—specific to NewsReader—listed in Table 2. The application of these criteria allows for a preliminary selection of relevant LOD datasets as candidates for partial inclusion in the background knowledge. These datasets are listed in Table 3. They are all open-licensed and their entities are well interconnected by owl:sameAs links, although they use different, largely incompatible vocabularies. A further selection is thus required to satisfy the *common vocabularies* criterion, and we decided (as in the first year) to restrict our focus to DBpedia, motivated by its use as a target for named entity linking as well as its central role in the LOD cloud.[61] In addition to DBpedia and according to the *TBox inclusion* criteria we also selected the definitions of the following vocabularies: DBpedia OWL ontology; Dublin Core elements (DC) and

[60] Statistics from `http://stats.lod2.eu/` as of December 2014. LOD statistics are highly time-depending and slightly inaccurate as influenced by the occasional inaccessibility of some datasets.

[61] Integrating other LOD datasets is currently not beneficial due to the lack of mappings between their vocabularies and the vocabulary of DBpedia. Only a few mappings are available between GeoNames feature types and DBpedia classes, but their application require the (expensive) use of an OWL reasoner, so we leave their use to a future revision of this work.

Table 2: Data selection criteria.

Criterion	Description
linkability	it must be possible for every entity gathered from LOD data to be linked to mentions in a news article, either directly from the named entity linking tool, or indirectly through a chain of owl:sameAs links
focus on entities	collected data should consist of entity descriptions as complete as possible; focus is on *real world* entities and not dynamically generated entities (e.g., the results of Web service) or metadata in general (e.g., Wikipedia page revisions for DBpedia entities)
data relevance	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, whereas inclusion of specialized datasets (e.g., MusicBrainz for music data) must be justified by the needs of specific use cases
focus on quality	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
common vocabularies	collected data should be expressed according to a common, properly designed vocabularies (e.g., Dublin Core, the DBpedia OWL ontology) that ease data querying and consumption
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 other considered vocabularies so to ease querying of data

terms (DCTERMS); Friend of a Friend (FOAF); Simple Knowledge Organization System (SKOS); WGS84 and GeoRSS[62] for geographic data.

DBpedia data is organized primarily by the language of the Wikipedia chapter it has been extracted from. Within NewsReader we are interested in both multilingual use cases with texts in the four project languages EN, ES, IT and NL, as well as single-language use cases with EN texts only. As the latter use cases would not benefit from a multi-language background knowledge, we decided to produce both an English (`en`) version and a multi-lingual (`ml`) version of the background knowledge, including respectively EN-only literals and literals in the four project languages. The first version maps to the selection of the EN DBpedia chapter, while the latter maps to the selection (and integration) of the EN, ES, IT and NL DBpedia chapters. It must be noted, however, that also other DBpedia chapters can contribute with relevant non-localized data, such as numeric and relational data. This happens for entities occurring in multiple DBpedia chapters, at least one referring to a language supported by NewsReader. In these cases, the description of the entity in the different chapters is not the same (in general, it is richer in the chapter of the language associated to the entity country) and it is thus possible to enrich our background knowledge with non-localized triples extracted from other DBpedia chapters. As this kind of enrichment may also introduce inconsistencies, we decided to produce two variants for each localization of the background knowledge (`en`, `ml`): one enriched with data from all the 18 available DBpedia chapters (`ext` variant) and the other one not enriched. Summing

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

Table 3: LOD datasets candidate for inclusion in the background knowledge.

Dataset	Description	Availability	Triples
DBpedia	Cross-domain dataset extracted automatically from 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. [Lehmann *et al.*, 2014] `http://dbpedia.org/`	RDF dump, SPARQL, URI dereferencing	1.98B
Freebase	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; to be closed and merged with Wikidata in mid 2015. [Bollacker *et al.*, 2008] `http://www.freebase.com/`	RDF dump, URI dereferencing	2.4B
YAGO2	Cross-domain dataset automatically extracted from Wikipedia, WordNet and GeoNames, with a rich type taxonomy (350K classes) and annotation of facts with confidence value, time and space validity. 95% accuracy manually measured. Linked to DBpedia. [Hoffart *et al.*, 2013] `http://www.mpi-inf.mpg.de/yago-naga/yago/`	RDF dump, URI dereferencing	120M
GeoNames	Geographic dataset of 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. `http://www.geonames.org/`	RDF dump, URI dereferencing	125M
Linked-GeoData	Geographic dataset automatically derived from OpenStreetMap with information about user-contributed points of interest (POIs) not covered by GeoNames. Linked to DBpedia and GeoNames. [Stadler *et al.*, 2012] `http://linkedgeodata.org`	RDF dump, REsT API	20B
Wikidata	Wikidata aims at becoming the central storage for structured data in Wikipedia and sister projects. It describes entities with property-value statements qualified with contextual data (e.g., time validity) and provenance *references*. Mapping of this data model to RDF is a work in progress and a research subject. [Erxleben *et al.*, 2014] `http://www.wikidata.org/`	unofficial RDF dump (see citation)	475M

up, we selected to generate the following four background knowledge datasets:

- `en` dataset, with EN-only literals and DBpedia EN data;
- `en_ext` dataset, with EN-only literals and DBpedia data for all languages;
- `ml` dataset, with EN, ES, IT, NL literals and EN, ES, IT, NL DBpedia data;
- `ml_ext` dataset, with EN, ES, IT, NL literals and DBpedia data for all languages.

For a specific language, DBpedia data is further divided based on topic. As not all topics are relevant for NewsReader, we applied again the criteria of Table 2 to narrow down the selection. As a result, the following DBpedia parts were selected for each language:[63]

- entity types and properties based on FOAF and the DBpedia vocabularies (files instance_types, instance_types_heuristic, mappingbased_properties_cleaned, persondata);
- entity names based on Wikipedia titles (file labels);
- entity categorization based on Wordnet 2.0 synsets (file wordnet_links);
- geographic coordinates of location entities (file geo_coordinates), keeping only property georss:point and dropping redundant data;
- links to entity images in Wikipedia (file images), including property foaf:depiction and excluding thumbnails data and copyright metadata (images are all open licensed);
- links to entity home pages (file homepages);
- brief language-dependent textual description of entities (file short_abstracts);
- owl:sameAs links among DBpedia chapters and among URIs and IRIs assigned to the same entity[64] (files interlanguage_links, iri_same_as_uri).

5.3.2 Data processing

Selected LOD files cannot be simply "concatenated" to produce the background knowledge datasets. Data must be filtered on a per-file basis, in order to remove unwanted data. Filtered data from different datasets must then be *smushed*,[65] 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 in triples describing the entity. Since an RDFS TBox is available, RDFS inference can be applied to derive implicit triples implied by TBox axioms and include them in the final datasets.

It is worth noting that smushing (also known as owl:sameAs inference) and RDFS inference are forms of reasoning that complement the specialized kind of reasoning on events

[63] Based on usage experience acquired in the first year, we excluded: (i) entity types based on YAGO2 and UMBEL (files yago_types, yago_taxonomy and umbel_links) and UMBEL vocabulary; (ii) entity categorization based on Wikipedia categories (files articles_categories, category_labels and skos_categories); (iii) links to Wikipedia pages and non home-page URLs (files external_links and wikipedia_links); (iv) mappings to schema.org and the Bibliographic Ontology, with corresponding vocabularies.

[64] 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.

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

```
# 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 19: Example of SPARQL query with (a) and without (b) smushing and inference.

described in Section 7. While the latter is specific to the way events are modeled in NewsReader, smushing and RDFS inference are standard forms of reasoning. Together, they greatly ease the consumption of stored data, as user queries can now assume that full knowledge about an entity (including implicit knowledge) is materialized in the KnowledgeStore and associated to a single, reference entity URI notwithstanding the many URI aliases the entity can have. This results in more simple and efficient queries, as shown in the example of Figure 19 where a simple query extracting names and surnames of persons is written both assuming (a) and not assuming (b) smushing and RDFS inference.

In order to perform the required data filtering, smushing and inference materialization, a processing pipeline has been assembled based on the RDF$_{\text{PRO}}$ tool described in Section 6. The pipeline automates these tasks based on a simple script and some configuration data specifying the URLs of the files to process and how to process them, customized for each of the four background knowledge datasets selected in Section 5.3.1. Both RDF$_{\text{PRO}}$ and the pipeline script and configuration data are published on the KnowledgeStore website, so that the whole process is repeatable and reconfigurable by anyone. The pipeline is organized in the four stages described next: *download*, *filtering*, *merging*, *analysis*.

Download stage Dataset and vocabulary files listed in the pipeline configuration are downloaded 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 by RDF$_{\text{PRO}}$ using a common format (Turtle Quads, i.e. N-Quads with Turtle encoding[66]) and compression method (gzip, due to good compression/speed tradeoff). Filtering is performed in a single pass on a per-triple basis. It allows us to drop triples with specific predicates and types on a per-file basis and, for every file, to remove literals in a language not supported and to

[66] `http://wiki.dbpedia.org/Internationalization/Guide`

rewrite blank nodes making them globally unique (this avoids clashes when merging data from multiple files). Triples in each filtered file are placed in a named graph associated to the DBpedia chapter it comes from, so to keep track of provenance in the next processing.

Merging stage This stage merges the filtered files previously generated, performing smushing and inference materialization and producing the final background knowledge dataset. Merging is implemented with RDF_{PRO} and requires three passes on filtered data:

- The first pass extracts TBox definitions that are stored in a TBox output file. TBox definitions are identified by searching for triples having certain terms from the RDF and RDFS vocabularies in the 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 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 URI 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; produced triples are then deduplicated and placed in a graph linked to the DBpedia chapters asserting or leading to those triples. It is worth noting that although RDF_{PRO} can produce the complete RDFS materialization of input data, we had to disable the inference rules (RDFS2 and RDFS3) on rdfs:domain and rdfs:range axioms, as many of them are imprecise and cause the materialization of a large number of 'incorrect' triples (e.g., that almost every dbo:place is also a dbo:person).

Analysis stage In this stage, a pass is done on (each) generated background knowledge dataset to collect a different number of statistics (see next section), which are stored to a file in form of VOID[67] [Alexander *et al.*, 2009] data enriched with a number of TBox concept annotations. These annotations and the TBox file previously extracted can be imported in tools such as Protégé enabling an easy navigation of extracted statistics.

5.3.3 Results

We configured and executed the pipeline in order to generate all the background knowledge datasets listed in Section 5.3.1: `en`, `en_ext`, `ml`, `ml_ext`. DBpedia version 3.9 was used initially, producing the datasets that were used during the second year of the NewsReader project. With the release of DBpedia 2014 in September, we reconfigured the pipeline and regenerated new versions of the datasets to be used in the next year. In the following,

[67] `http://www.w3.org/TR/void/`

Table 4: Number of triples in produced datasets.

Dataset	rdf:type	owl:sameAs	ABox (other)	TBox	Total
en_39	11 583 289	783 200	49 312 382	15 039	61 693 910
en_ext_39	13 239 452	783 200	61 153 352	15 041	75 191 045
ml_39	16 142 863	4 239 891	76 073 537	15 043	96 471 334
ml_ext_39	16 917 881	4 239 891	83 220 239	15 043	104 393 054
en_2014	15 184 293	859 158	59 774 033	19 798	75 837 282
en_ext_2014	17 279 671	859 158	77 751 702	19 816	95 910 347
ml_2014	23 881 931	4 691 477	94 079 675	19 810	122 672 893
ml_ext_2014	24 872 604	4 691 477	104 800 908	19 816	134 384 805

Table 5: Number of entities in produced datasets.

Dataset	PER	ORG	GPELOC	FAC	PROD	WOA	EVENT	MISC	TOTAL
en_39	1 124 450	329 693	621 437	144 143	119 734	372 953	70 464	1 270 659	4 049 227
en_ext_39	1 246 076	344 802	666 932	165 266	123 814	399 611	94 049	1 292 297	4 299 465
ml_39	1 368 929	350 210	748 181	273 780	132 853	462 536	112 982	1 910 307	5 342 574
ml_ext_39	1 427 713	362 106	764 098	285 686	135 415	482 721	123 790	1 945 551	5 493 613
en_2014	1 649 672	302 727	849 711	164 646	128 196	389 118	89 216	1 226 617	4 634 402
en_ext_2014	1 673 873	324 190	915 939	187 701	133 324	424 830	115 921	1 306 946	4 863 236
ml_2014	2 069 158	352 411	1 275 738	423 843	144 297	499 760	220 940	2 056 611	6 606 109
ml_ext_2014	2 076 553	366 079	1 310 817	434 989	147 350	525 553	232 569	2 111 575	6 738 541

we provide some statistics about the processing done and the resulting datasets, covering both the versions based on DBpedia 3.9 (tagged with _39) and the ones based on DBpedia 2014 (tagged with _2014) so that a comparison between the two can be made.

Table 4 reports the number of triples in each dataset, distinguishing between rdf:type triples, owl:sameAs triples, other ABox triples (essentially expressing entities properties) and TBox triples. Moving from DBpedia 3.9 to DBpedia 2014 causes an increase of dataset size of $23 - 28\%$. Enrichment with data from other DBpedia chapters causes an increase in size of $22 - 27\%$ for the `en` dataset, and $8 - 10\%$ for the `ml` datset. Multi-lingual (`ml`) datasets use more owl:sameAs triples to link canonical entity URIs to their aliases in the integrated DBpedia chapters. The TBox size is largely independent of the considered dataset.

Table 5 reports the number of entities[68] in each dataset, divided based on the types used in the (revised) annotation guidelines of WP3, i.e.: persons (PER), organizations (ORG), geo-political entities and locations (GPELOC), facilities (FAC), products (PRO), works of art (WOA) and events (EVENT), with OTHER representing DBpedia entities that could not be classified under previous types.[69] As the classification of entities is not

[68]Entities 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.

[69]Classification according to the types of the annotation guidelines has been performed on the basis of

Table 6: Processing statistics.

Dataset	Files	Output triples				Execution time [s]			
		Download	Filtering	Merging	Analysis	Filtering	Merging	Analysis	Total
en_39	17	77 770 116	70 268 829	61 693 943	47 020	1288	601	250	2139
en_ext_39	85	139 812 619	110 394 636	75 191 078	131 938	1793	715	321	2829
ml_39	43	178 654 555	120 011 745	96 471 381	73 643	2284	900	392	3576
ml_ext_39	98	219 324 567	145 968 914	104 393 101	137 944	2767	1028	454	4249
en_2014	17	98 897 879	82 371 053	75 837 313	49 051	1269	848	372	2489
en_ext_2014	85	176 899 005	133 811 216	95 910 378	168 962	2225	1043	398	3666
ml_2014	43	236 496 207	144 612 632	122 672 937	82 822	2975	1077	526	4578
ml_ext_2014	98	287 518 712	177 634 142	134 384 849	171 677	3584	1349	577	5510

exclusive in DBpedia, the total value does not represent the sum over the individual entity types.

Table 6 reports some statistics on the processing performed with the pipeline for each dataset: number of input files; number of triples resulting from each stage of the pipeline; and execution times of each stage and of the pipeline as a whole. Times were measured on a RedHat 6.4 (Linux 2.6) workstation with an Intel(R) Core(TM) i7 CPU, 16 GB RAM and 500 GB disk. On average, throughputs are 78K triples/s for the filtering stage, 130K triples/s for the merging stage and 233K triples/s for the analysis stage. Filtering is slow as it operates on DBpedia data compressed with `bzip2`. We omit the execution times of the download stage as its performances depends on bandwidth availability.

We conclude pointing out that produced datasets and their statistics are available on the KnowledgeStore website. In particular, statistics of a dataset are distributed as an annotated *statistics ontology* in two versions: a full version covering all concepts and a more compact (and manageable) version having only concepts with more than 100 instances. This ontology can be imported in tools for ontology editing and browsing such as Protégé, as shown in Figure 20, and can help in understanding and using the dataset, e.g., by supporting the construction of SPARQL queries.

5.4 The KnowledgeStore in action: use cases

The KnowledgeStore was successfully deployed, populated, and exploited to build enhanced applications in two concrete NewsReader scenarios, described next. In particular, we will

the DBpedia classes (dbo: namespace) associated to entities using the following mapping: Person → PER; Organisation → ORG; NaturalPlace, PopulatedPlace, ProtectedArea, SiteOfSpecialScientificInterest, WineRegion, FrenchSettlement, CelestialBody, WorldHeritageSite, Mountain, HistoricPlace, Community, CountrySeat → GPELOC; Monastery, Monument, SportFacility, ArchitecturalStructure, SkiArea, Park, Garden, Cemetery, Abbey → FAC; Database, Document, Software, Website, Device, Drug, Flag, Food, Aircraft, Automobile, Locomotive, Rocket, Ship, Train → PROD; Artwork, Cartoon, CollectionOfValuables, Film, LineOfFashion, Musical, MusicalWork, RadioProgram, TelevisionEpisode, TelevisionSeason, TelevisionShow, WrittenWork → WOA; Event, Holiday, Award, Sales, SportsSeason → EVENT.

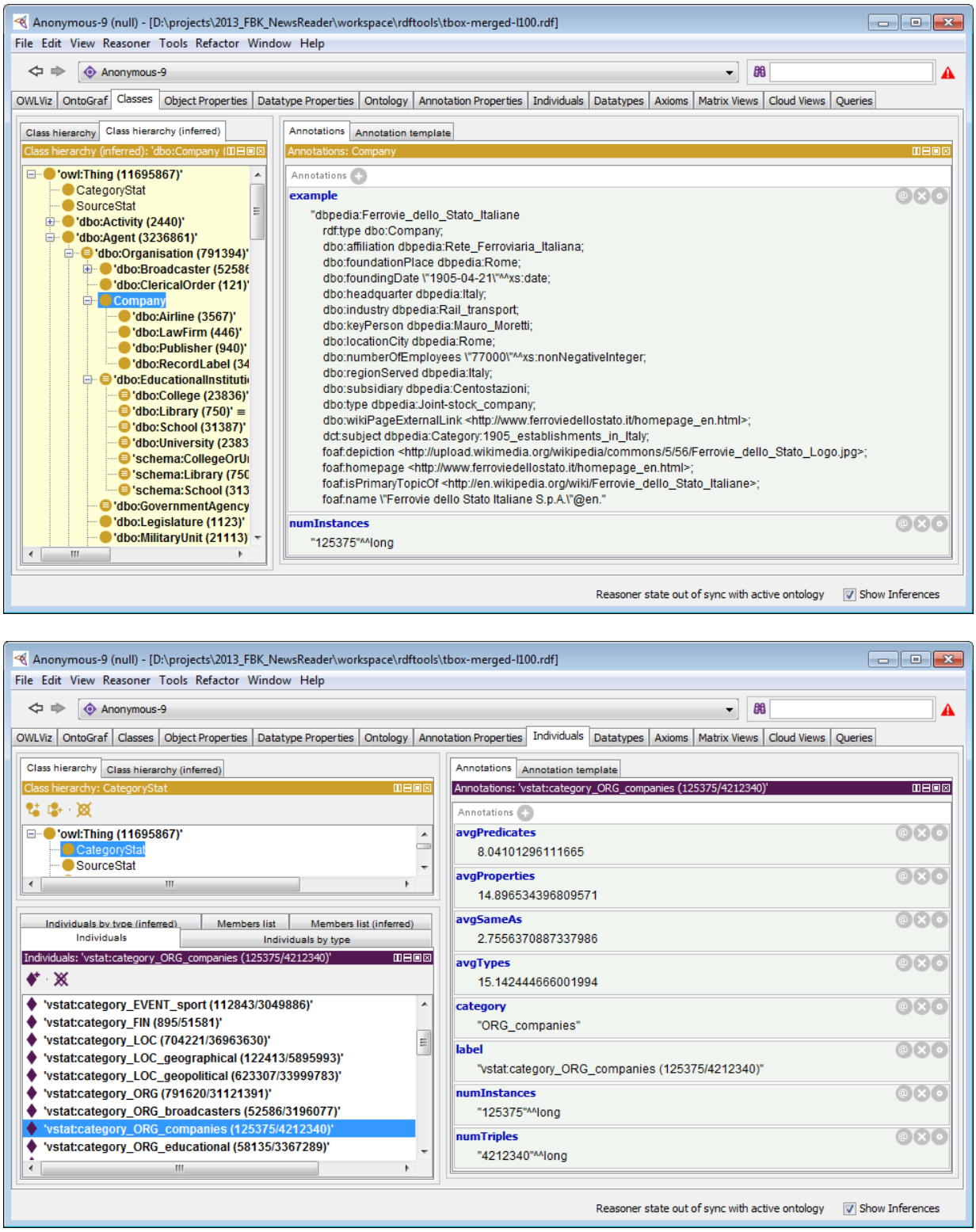


Figure 20: Examples of browsing the statistics ontology in Protégé.

introduce the KnowledgeStore clients involved in the project, as well as some statistics on the resources, mentions, and entities handled in each scenario.

5.4.1 Scenario 1: Global Automotive Industry (version 1)

The first scenario is about building a decision support tool to analyze the news related to the last decade financial crisis, with a special focus on the global automotive industry sector, in order to mine its key events, and to understand the role of major players (e.g., CEOs, companies) in it.

Clients: Information extraction processors Two main processors were actually involved: a single-resource processor, the *NewsReader NLP pipeline*,[70] and a cross-resource processor, the *VUA Event Coreference Module*.[71]

The NewsReader NLP pipeline processes each news document provided in input, annotating it according to the NLP Annotation Format ([Fokkens *et al.*, 2014], NAF) with information about: tokenization, lemmatization, part-of-speech tagging, parsing, word sense disambiguation, named entity linking, semantic role labelling, nominal coreference, temporal expression recognition, opinion mining, and event coreference. The output of the NewsReader NLP pipeline is fed into the KnowledgeStore, populating the resource and mention layers.

The VUA Event Coreference Module works on the results produced by the NewsReader NLP pipeline by processing a collection of news. It instantiates the entities (e.g., events, persons, organizations), and assertions on them, that corresponds to the mentions extracted by the NewsReader NLP pipeline on each document, trying to understand whether two different mentions, potentially extracted from different news, actually refer to the same entity.

Clients: Populators Two populators were developed. The *NAF populator* is used to upload the news documents into the KnowledgeStore resource layer, setting the value of several metadata attached to the document (e.g., publication date, author, title). It is also invoked at the end of the NewsReader NLP pipeline to upload in the KnowledgeStore the complete NAF annotated version of the source news (in the resource layers), and to inject in the KnowledgeStore the mentions (and their metadata) extracted by processing the news.

The structured content populator, called RDF_{PRO}[72] [Corcoglioniti *et al.*, 2014b] is used to populate the KnowledgeStore with *background knowledge*, i.e., RDF content directly injected into the KnowledgeStore entity layer, that may (i) support some of the tasks performed by the information extraction processors, and (ii) complement the information automatically extracted from news with quality content available in structured resources such DBpedia, Freebase, and Geonames, to favour the exploitation of the KnowledgeStore

[70] `http://ixa2.si.ehu.es/nrdemo/demo.php`
[71] `http://ic.vupr.nl/~ruben/vua-eventcoreference.ttl/`
[72] `http://fracor.bitbucket.org/rdfpro/`

content by applications built on top of the KnowledgeStore. It is also used to inject into the KnowledgeStore the entities and axioms produced by the VUA Event Coreference Module.

Clients: Applications The main application in this scenario is *SynerScope*,[73] a visual analytics application that delivers real time interaction with dynamic network-centric data. Synerscope interacts with the KnowledgeStore through the *KnowledgeStore exporter* [van Hage and Ploeger, 2014], a tool that converts the data stored in the KnowledgeStore in the format digested by SynerScope. SynerScope offers different views (e.g., table view, hierarchical view, map view) on the KnowledgeStore content, enabling users to navigate it through various interaction methods (e.g., selection/highlight, drill down/up, expansion). This way, it is possible to visually browse all events that involve a given person or company, or to build networks of persons/companies based on event co-participation.
Besides SynerScope application, the capability to query the KnowledgeStore content favours the delivery of automatically generated reports (and plots) supporting decision makers. For instance, retrieving the different events involving the ten major car companies, it was possible to generate a report showing the trend of the quantity of events per year in which these companies were involved in the considered period, and therefore to assess their popularity (according to the considered dataset) during the economical crisis. Similarly, retrieving the different events and the locations and times they took place at, we were able to produce maps (one per year) to obtain insights into how the localities of the global automotive industry changed during the crisis.

Statistics on the populated KnowledgeStore 63.635 news documents from various providers distributed over a time period of ten years (2004-2013)[74] were processed and uploaded into the KnowledgeStore. The NewsReader NLP pipeline produced more than 8 millions mentions processing them, while the VUA Event Coreference Module instantiated 2.2 millions of entities (including, ~198K persons, ~185K organizations, ~36K locations, and ~1.8M events) and approximately 33 millions of triples about them.[75] The entity layer of the KnowledgeStore was also populated with 270 millions of statements coming from a selected multi-lingual subset of DBpedia.[76] Roughly 30 hours were needed to populate the whole system from scratch (~1.6 seconds per news).

5.4.2 Scenario 2: FIFA 2014 World Cup

The second scenario is about building web-based applications to reveal hidden facts and people networks behind the FIFA World Cup (2014). While data collection and preparation required significant time and effort, the development of applications on top of stored con-

[73] `http://www.synerscope.com/`

[74] Made available for project purposes by LexisNexis - `www.lexisnexis.nl/`

[75] RDF data produced by the VUA Event Coreference Module is publicly available at `http://datahub.io/dataset/global-automotive-industry-news`

[76] The background knowledge dataset used is available for download at `http://knowledgestore.fbk.eu`.

tents was realized as part of a Hack Day event,[77] where 40 people, a mixture of linked data enthusiasts and data journalists, gathered for one day to collaboratively develop web-based applications on top of the KnowledgeStore.

Clients: Applications[78] Ten web-based applications, implemented in different programming languages, were developed on top of the KnowledgeStore in roughly 6 working hours. The development was facilitated, especially for people not familiar with Semantic Web technologies such as RDF and SPARQL, by the availability of a python-based API, the NewsReader Simple API[79] [Hopkinson *et al.*, 2014], where each method implements a SPARQL query template instantiated at runtime with the actual parameters passed to the method (e.g., the method "Actors of a specified type" implements a query that returns all instances having as RDF type the value of the parameter passed to the method). Each application was developed with a focused purpose, among them: to determine which team a named football player had played during his career (by looking at transfer events); to discover which football teams were most commonly associated with violence; to determine people and companies related to gambling; and, to establish the popularity of people, companies, and football teams in different locations.

Statistics on the populated KnowledgeStore 212,258 football-related news documents, from various providers (including BBC Sport and The Guardian web-sites) and distributed over a time period of ten years (2005-2014), were processed and uploaded into the KnowledgeStore. The NewsReader NLP pipeline produced more than 72 millions of mentions processing them, while the VUA Event Coreference Module instantiated 10.2 millions of entities (including, ~402K persons, ~427K organisations, ~32K locations, and ~9.3M events) and approximately 136 millions of triples about them. The entity layer of the KnowledgeStore was also populated with 104 millions of statements coming from a selected subset of DBpedia.[80] Roughly 56 hours were needed to populate the whole system from scratch (~0.9 seconds per news). During the Hack Day, the KnowledgeStore received 30,126 queries (on average, 1 query/second, with peaks of 20 queries/second), issued either directly trough the SPARQL endpoint or via the NewsReader Simple API, and successfully served them on average in 654ms (only 40 queries out of 30,126 took more than 60 seconds to complete).

We conclude with some technical specifications of the cluster of machines that hosted the KnowledgeStore in both scenarios: five server machines were used (one for Virtuoso and the KnowledgeStore frontend, four machines for the HBase+Hadoop environment), equipped with 32GB of RAM, 12 core CPUs, and running a RedHat Enterprise Linux OS.

[77] `http://www.newsreader-project.eu/newsreader-world-cup-hack-day/`, held in London, 10th of June 2014.

[78] The KnowledgeStore data model, information extraction processors, and populators exploited to inject content in the KnowledgeStore were the same (or minor variants) of the ones described in Scenario 1.

[79] Accessible at: `https://newsreader.scraperwiki.com`, code available at `https://bitbucket.org/scraperwikids/newsreader_api_flask_app`

[80] The background knowledge dataset used is available for download at `http://knowledgestore.fbk.eu`.

5.4.3 Scenario 3: Global Automotive Industry (version 2)

The third scenario involves a set of news related to the same topic as in Section 5.4.1: one decade of financial crysis, focused on the automotive industry sectore. There are some major differences from the first scenario, such as the number of articles, the processors used, and an additional reasoning part.

Clients: Information extraction processors For this resource, a new version (2.1) of the *NewsReader NLP pipeline*, and the cross-resource processor, the *VUA Event Coreference Module* have been used. In addition to the modules described in Section 5.4.1, for the first time the ESO reasoner has beed applied (Section 7) to get new information inferred from the events extracted by the pipeline.

Clients: Applications The main applications of this dataset are the two 2015 hackhaton (January 21st in Amsterdam[81]) and January 30th in London[82]). The events were of interests for data journalists on an automotive desk, analysts sifting daily news looking for information on a particular company or on competitors, data analysts looking to understand how customers operate their supply chain, analysts trying to find secondary events that could influence an investment decision.

Statistics on the populated KnowledgeStore A total of 1,259,748 automotive industry-related news (extracted from various sources and distributed in the last 10 years) were processed and used to populate the KnowledgeStore. The NLP pipeline produced more than 200 millions of mentions, resulting in around 100 millions of events extracted by the VUA Event Coreference Module, linking to DBpedia ~652K persons, ~873K organisations and ~274K locations. Using the ESO reasoner 7, a total of 327,644 situations have been extracted (168,848 as *pre-situation*, 152,492 as *post-situation* and 6,304 as *during-situation*). The triple store of the KnowledgeStore contains 450M of triples: 350M from the VUA Event Coreference Module, 96M from the background knowledge, 2M from the ESO reasoner. Roughly a couple of weeks were needed to populate the whole system from scratch (1 second per news article).

5.5 Discussion

Nine months of usage, user feedbacks and improvements of the KnowledgeStore within NewsReader have provided us with valuable insight on the practical issues and the user expectations encountered when deploying a system like the KnowledgeStore, permitting us to validate our design and identify its weaknesses. In this section we discuss our findings, that we believe are of general interest for any system covering the KnowledgeStore role.

Population throughput A loading time of 1.6 seconds per news article (Section 5.4.1, then lowered to 0.9 seconds and further improvable) may appear inadequate, although it must be noted that it comprises the indexing of two resources—the news and its annotation—and possibly of a few thousand of mentions, for a total of several MBs of data. The NLP

[81] `http://www.newsreader-project.eu/hackathon-newsreader-amsterdam-jan-21-2015/`
[82] `http://www.newsreader-project.eu/hackathon-newsreader-london-jan-30-2015/`

processing required to produce these annotations and mentions is sensibly slower and makes the reported time negligible and its optimization ineffective at a global level.

Read/write separation When designing the KnowledgeStore, we targeted the scenario where a stream of data is continuously fed into the system (e.g., daily news and data extracted from them), resulting in the concurrent access from multiple clients with a mixed read/write load. However, practical experience has shown a sharp separation between read and write accesses, with populators and information extraction processors performing large, infrequent write operations, whereas access from applications is essentially read-only; in the extreme–but relevant–case where data does not change in time, this pattern can result in a *write once, read many* behaviour. This evidence opens the possibility for a number of architectural optimizations. In particular, it suggests the use of multiple storages with (asynchronous) *master-slave replication*, where writes are targeted at a master storage which is then replicated to (one or more) slave storages serving the read-only load.

Unified query language As motivated in Section 4, the choice of a triple store and SPARQL for managing (schema-free) entity data and of a more-scalable store and a simpler query language for managing (essentially relational) resource and mention data appears natural for a system like the KnowledgeStore, as it brings a number of benefits in terms of scalability and compatibility with Semantic Web best practices. Concrete usage of the system shows that users appreciate the expressivity of SPARQL but, deeming inadequate the query language of the CRUD endpoint, they ask for a unified, SPARQL-like language targeted at all the contents of the KnowledgeStore. Providing such a language is however a challenging task (possibly a research problem on its own) due to the volume of data and the different storage backends involved. While we are investigating this direction, we are also considering a more drastic shift from SPARQL (initially considered a requirement) to another *graph query language*, using a single graph database as a unified storage backend for all the types of data managed by the KnowledgeStore. While this change will free us from the limits of SPARQL and current triple store technology, it will also mark a partial departure from Semantic Web standards and its implications and user acceptance have to be carefully evaluated.

Analytical queries We expected SPARQL queries to be quite selective and affect only limited portions of entity data (e.g., 'retrieve all the information about a certain entity'), but logs exhibit many analytical SPARQL queries that match and/or extract a sensible amount of data stored in the KnowledgeStore. It turns out that users submit SPARQL queries to compute statistics, to analyze the loaded corpus and to assess the results and performances of information extraction processors. While SPARQL can be used to a certain extent for analytical purposes, there are many cases where these queries take very long times to execute or even fail, e.g., due to out-of-memory (often due to improper query planning) or timeout at the API level (as the synchronous message passing scheme we use is not suited to long running operations). This kind of load is better served using parallel and distributed approaches for bulk data processing, such as MapReduce or derivations of

the *Bulk Synchronous Parallel* (BSP)[83] paradigm, possibly together with specification languages that users can exploit to formulate their analysis, such as the procedural data-flow language Pig Latin[84] or the declarative, SQL-based language HiveQL.[85] Analysis formulated in this way can be registered by users and precomputed on a periodic basis or when data changes, so that their results are always readily available.

Flexible access control Access control becomes a requirement in presence of copyrighted content whose provision and consumption involve different parties. This aspect turns out to be particularly relevant in research scenarios (such as NewsReader), where dissemination needs conflict with the need of content providers to protect their intellectual property. In general, different access control policies apply to resources from different sources and, within a resource, to its text and various metadata attributes (e.g., title and data can be publicly accessible whereas author and text not). Access control policies also apply to mention and entity data derived from copyrighted resources, with the situation being more complex for entity data as it combines information extracted from multiple resources, possibly with different distribution policies.[86] While we anticipated the need for an access control mechanism in the KnowledgeStore, we had to revise it several times during the use of the system in order to accommodate unanticipated requirements. Therefore, we stress the importance for systems like the KnowledgeStore of a flexible access control mechanism able to accommodate known requirements and cope (as far as possible) with their unanticipated change in time.

Tighter integration with information extraction pipeline Although integrating an information extraction pipeline is not an expensive activity and can benefit from a number of readily available NLP tools, it still require a good knowledge of NLP concepts, tools and best practices. This hinders a wider usage of the KnowledgeStore by users that do not have this kind of background. For that reason, we are investigating the possibility of defining an extension point in the KnowledgeStore where standardized, possibly pre-packaged and pre-configured NLP pipelines can be plugged in and automatically invoked by the KnowledgeStore when a resource is uploaded in the system. This would allow casual users to start with a standard pipeline and immediately have a running system. At the same time, advanced users will be still able to assemble their pipeline and plug it in the system.

Scaling down the system While a system like the KnowledgeStore should be designed with massive scalability and deployment on a distributed infrastructure in mind, in practice we encountered a number of usage scenarios that do not require scalability and instead mandate for simple, lightweight single-machine deployments; these scenarios include the use of the system for evaluation or demonstration purposes and any other use involving

[83] `en.wikipedia.org/wiki/Bulk_synchronous_parallel`

[84] `http://pig.apache.org/`

[85] `https://hive.apache.org/`

[86] Protecting entity data extracted from resources such as news may seem unnecessary, as it usually convey public domain facts. Still, extraction may be imprecise and content providers may wish not to be held responsible for errors in extracted data in case it is published.

small datasets. It must be noted that the storage backends required in the two deployment situations are very different. A setup for massive scalability employs distributed software infrastructures (e.g., HBase) with sensible overheads that require multiple machines to be competitive,[87] whereas a lightweight setup uses simpler components (e.g., an embedded relational database) that cannot handle large data sizes. Therefore, a flexible architecture with alternative, pluggable storage backends is crucial to enable the system to *scale up* but also to *scale down* and be used in a multitude of deployment scenarios. We fortunately chose an extensible, plugin-based architecture for the KnowledgeStore, and we are currently implementing lightweight replacements of the various storage plugins to enable single-process, single-machine deployments of the KnowledgeStore.

[87] The minimal KnowledgeStore setup with Hadoop, HBase and Virtuoso requires 7 processes that can hardly run on a constrained machine such as a laptop.

6 The RDF$_{PRO}$ Tool

Changes wrt the KnowledgeStore Deliverable D6.2.1

- Brand new section.

In this section, we consider the feasibility of processing billions of RDF triples on a single commodity machine using streaming and sorting techniques and focusing on RDF processing tasks relevant for Linked Data consumption: data filtering and transformation, RDFS inference, owl:sameAs smushing and statistics extraction. To investigate this research question we built RDF$_{PRO}$ (RDF Processor), an open source tool that provides streaming and sorting-based processors for the considered tasks and allows for their sequential and parallel composition in complex pipelines. An empirical evaluation of RDF$_{PRO}$ in four application scenarios — dataset analysis, filtering, merging and massaging — shows the effectiveness of the tool and allows us to positively answer our research question. We present RDF$_{PRO}$ model in Section 6.1, its processors in Section 6.2 and its implementation in Section 6.3.

6.1 Processing model

The processing model of RDF$_{PRO}$ is centered around the concept of *RDF processor*. A processor @P[88] (Figure 21a) is a software component that consumes an *input stream* of RDF quads—i.e., RDF triples with an *optional* fourth named graph component[89]—in *one or more passes*, produces an *output stream* of quads and may have an internal state as well as side effects like writing or uploading RDF data.

Streaming characterizes the way quads are processed: one at a time, with no possibility for the processor to "go back" in the input stream and recover previously seen quads. Specifically, a processor declares the number $n \geq 1$ of passes it needs, and may be asked to perform $m \geq n$ passes on its input (e.g., to support multiple downstream passes). In the first $n-1$ passes (if any), the processor reads the input and updates its state. At pass n the input is read again and output quads are emitted for the first time. In the next $m-n$ passes (if any), the processor receives the input again and must emit the same quads of pass n (the order may change).

Sorting is offered to processors as a primitive to arbitrarily sort selected data—possibly (a subset of) input quads—during a pass. Sorting is often combined to streaming in the literature as it overcomes many of the limitations of a pure streaming model [Aggarwal *et al.*, 2004; O'Connell, 2009]. In particular, it enables duplicates removal and set operations and provides the capability to group together information that may be scattered in the stream but must be processed together (e.g., all the quads about an entity when computing statistics). At the same time, most platforms provide an highly-optimized sorting utility that fully exploits available hardware resources: multiple CPU cores to parallelize the

[88] Processors are denoted by '@' in RDF$_{PRO}$ syntax.

[89] The graph component is unspecified for triples in the *default graph* of the RDF dataset (see RDF 1.1 and SPARQL specifications); this allows for using RDF$_{PRO}$ on plain triple data.

sorting algorithm, disk space to manage large datasets via (disk-based) *external sorting* and memory space to speed up processing.

Starting from the processors supplied with RDF$_{\text{PRO}}$ (Section 6.2) or implemented by users, new *pipeline* processors can be derived by (recursively) applying sequential and parallel compositions. In a *sequential composition* (Figure 21b), two or more processors @P$_i$ are chained so that the output stream of @P$_i$ becomes the input stream of @P$_{i+1}$. In a *parallel composition* (Figure 21c), the input stream is sent concurrently to several processors @P$_i$, whose output streams are merged into a resulting stream using one of several possible *set operators* (specified with a flag `f` in figure and syntax): *union with duplicates* (flag `a`), *union without duplicates* (`u`), *intersection* (`i`) and *difference* (`d`) of quads from different branches. The number and orchestration of passes resulting from composition are automatically managed.

An example of composition is shown in Figure 21d: a Turtle+gzip file (`file.ttl.gz`) is read, TBox and VOID [Alexander *et al.*, 2009] statistics are extracted in parallel and their union is written to an RDF/XML file (`onto.rdf`). Notably, I/O in the example do not use the input and output streams of the pipeline processor (dotted box in the figure), but rely on specific @write and @read processors whose side effects are dumping and augmenting the stream with the content of external files. These I/O processors provide a lot of flexibility in how data is read and written, as they can be placed at any point of a pipeline removing the limit of single input and output streams (indeed, the RDF$_{\text{PRO}}$ tool relies on these processors for all I/O, ignoring global input and output streams that are instead accessible when using RDF$_{\text{PRO}}$ as a library).

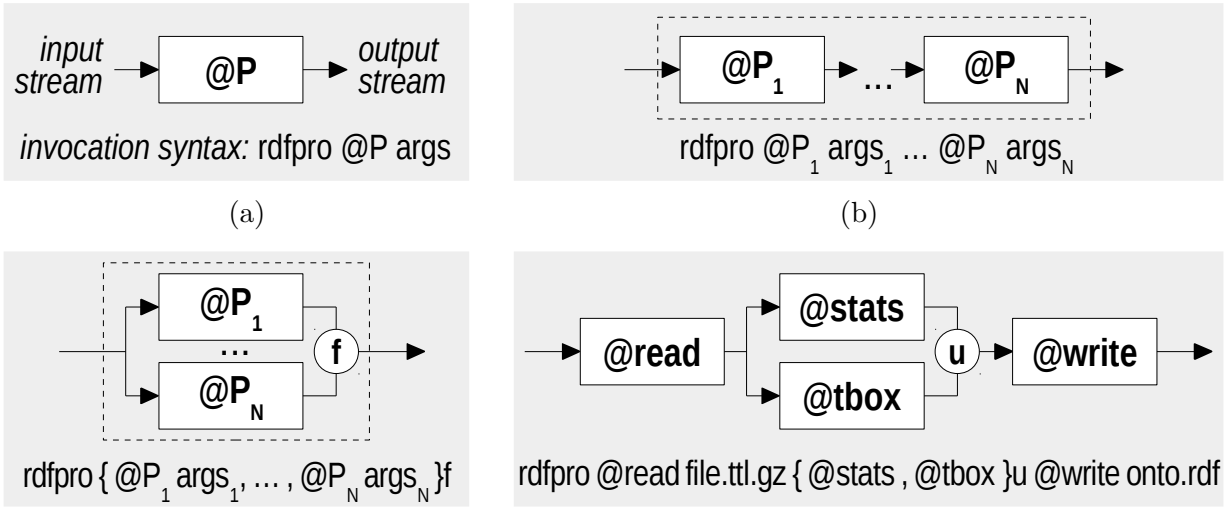


Figure 21: Processor (a); sequential (b) & parallel (c) composition; example (d) – full syntax on web site.

6.2 Processors

In order to address the processing tasks considered at the beginning of this section, we implemented the following processors in RDF$_{\text{PRO}}$:

@read Reads RDF file(s), emitting their quads together with the input stream; rewrites bnodes to avoid clashes.

@write Writes quads to one RDF file or splits them to multiple files evenly; quads are also propagated in output.

@download Sends a query to a SPARQL endpoint to download quads that are emitted by augmenting the input stream. Both CONSTRUCT and SELECT queries are supported: the first can only return triples in the default graph; the latter produces bindings for specific variables `s`, `p`, `o`, `c` that are used to build output quads.

@upload Uploads quads from the input stream to an RDF store using SPARQL INSERT DATA calls, in chunks of a specified size; quads are also propagated in output.

@transform Processes each input quad with a user-supplied Groovy[90] script that can either discard the quad, propagate it or transform it into one or more output quads.

@smush Performs *smushing*, using a ranked namespace list to select canonical URIs that are linked in output to coreferring URIs (aliases) via owl:sameAs quads.

@infer Computes the RDFS deductive closure of its input. The TBox, read from a file, is closed and emitted first. Domain, range, sub-class and sub-property axioms are then used to do inference on input quads one at a time, placing inferences in the same graph of the input quad.[91] Specific RDFS rules may be optionally disabled to avoid unwanted inferences.

@tbox Filters the input stream by emitting only quads of TBox axioms. Both RDFS and OWL axioms are extracted, even if the latter are not used by @infer.

@stats Emits VOID structural statistics for its input. A VOID dataset is associated to the whole input and to each *source* URI linked to named graphs in the data by a configurable property; class and property partitions are produced for each dataset. Additional terms extend VOID to express the number of TBox, ABox, rdf:type and owl:sameAs quads, the average number of properties per entity and informative labels and examples for TBox terms, viewable in tools such as Protégé.

@unique Discards duplicates in the input stream. Optionally, it merges quads with same subject, predicate and object but different graphs in a unique quad. To track provenance, this quad is placed in a graph inheriting the descriptions of source graphs (i.e., the quads having them as subject) and representing their "fusion".

[90] Groovy is a scripting language well integrated with Java and reusing its libraries. See `http://groovy.codehaus.org/`

[91] This scheme avoids expensive join operations and works with arbitrarily large datasets whose TBox fits into memory. Inference is complete if: (i) domain, range, sub-class and sub-property axioms in the input stream are also in the TBox; and (ii) the TBox has no quad matching patterns:

- X rdfs:subPropertyOf {rdfs:subClassOf | rdfs:domain | rdfs:range | rdfs:subPropertyOf}
- X {rdf:type | rdfs:domain | rdfs:range | rdfs:subClassOf} {rdfs:Datatype | rdfs:ContainerMembershipProperty}

6.3 Implementation

RDF$_{\text{PRO}}$ is implemented in Java on top of the open source Sesame RDF library.[92] It consists of a *runtime* where multiple *processor implementations* can be plugged in, assembled using sequential and parallel composition and executed.

Runtime implementation The runtime defines the API of RDF processors and manages their lifecycle. Processors are Java classes extending `RDFProcessor` and declaring the number of passes they need. Each processor is attached to an input *quad queue* and an output *quad sink*. Input quads from the queue are "pushed" to the processor by invoking a callback method, using multiple threads from a common pool to process quads in parallel; other callbacks are invoked at the beginning and end of each pass to allow for initialization and completion of stateful computations. Output quads are emitted to the quad sink (a Sesame `RDFHandler`), with the runtime taking care of their downstream processing (if any). The design is inspired to the *Staged Event Driven Architecture* (SEDA) [Welsh *et al.*, 2001], with processors playing a passive role and all the queue and thread management handled by the runtime with the goal of maximizing CPU usage.

Within the runtime, streaming is embodied in the `RDFProcessor` API and in the management of input and output streams. Sorting, instead, is realized as a reusable primitive that can be invoked by the runtime and by processor implementations. This primitive is realized on top of the native, highly-optimized `sort` Unix utility, using *dictionary encoding techniques*[93] to compactly encode frequent RDF terms in sorted data, reducing its size (we measured ∼40 bytes per quad on real-world data) and improving execution times at the price of some memory consumption for the dictionary.

Processor composition is also managed by the runtime. Sequential composition and union with duplicates are computationally cheap, while the other forms of parallel composition are more expensive due to their use of sorting. In particular, intersection and difference are implemented by appending a label identifying the source branch to each quad sent to `sort`, and then gathering and checking all the labels of a sorted quad to decide if it can be emitted.

Processor implementation Due to their central role, the @read and @write processors feature multi-thread implementations aiming at transferring data as fast as possible to avoid I/O bottlenecks. Multiple RDF files can be parsed and written in parallel and, for line-oriented RDF formats, a single file can be split in newline-terminated chunks that are processed concurrently to increase the data throughput.

The @smush processor performs two passes: the first to extract the owl:sameAs graph which is kept in memory; the second to replace URIs based on detected equivalence classes. Efficient memory consumption is achieved with a specialized data structure that uses a custom hash table with an open addressing scheme to index URIs; table entries contain also a *next pointer* that organizes URIs of an owl:sameAs equivalence class in a circular linked

[92] `http://www.openrdf.org/`

[93] We encode TBox URIs of known vocabularies (from `prefix.cc`) with integers. Namespaces and local names of other URIs are separately encoded until the encoding tables are full, after which they are emitted unchanged. For literals we encode the language and datatype tags, but not their labels.

list, which expands as new owl:sameAs quads are encountered and allows the structure to grow linearly with the number of URI aliases.

The @infer processor performs TBox inference using an in-memory, *semi-naive forward-chaining* algorithm [Ceri *et al.*, 1989]. ABox inference is done one quad at a time, using multiple threads and special deduplication logic and data structures for removing as many duplicate inferred quads as possible, so to avoid an artificial "explosion" of the number of output quads.

The @stats processor is implemented by sorting the input stream twice (simultaneously within a single pass): based on the subject, to group quads about the same entity and compute entity-based and distinct subjects statistics; and based on the object, to compute distinct objects statistics. Partial statistics are kept in memory during the processing.

6.4 Empirical Evaluation

To answer our research question, we perform an empirical evaluation of RDF$_{\text{PRO}}$ in four broad, relevant usage scenarios that exemplify the considered RDF processing tasks. In the first three scenarios—*dataset analysis* (Section 6.4.1), *filtering* (Section 6.4.2) and *merging* (Section 6.4.3)—we conduct practical experiments using a commodity workstation[94] and popular datasets (Freebase, DBpedia, GeoNames) whose contents and sizes are representative of the ones typically encountered by LOD applications. In the fourth scenario—*dataset massaging* (Section 6.4.4)—we categorize miscellaneous data massaging tasks that can be addressed with our approach and show its larger applicability; due to the simple processing involved we do not conduct experiments here. An extended description of the scenarios, including scripts for reproducing the experiments, is reported on RDF$_{\text{PRO}}$ website.[95]

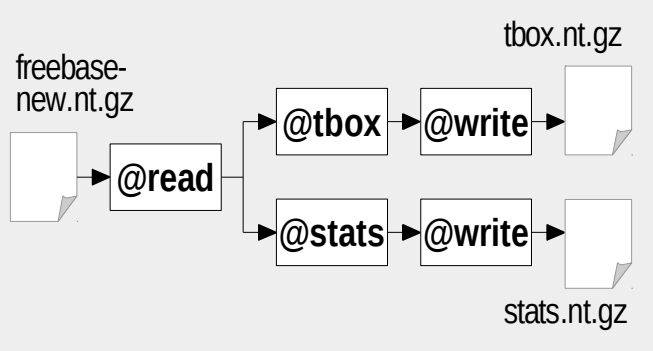
6.4.1 Dataset Analysis

Dataset analysis comprises all the tasks aimed at providing a qualitative and quantitative characterization of the contents of an RDF dataset, such as the extraction of the data TBox or of instance-level ABox data statistics (e.g., VOID). When processing RDF, dataset analysis can be applied both to input and output data. In the first case, it helps identifying relevant data and required pre-processing tasks, especially when the dataset scope is broad (as occurs with many LOD datasets) or its documentation is poor. In the second case, it provides a characterization of output data that is useful for validation and documentation purposes.

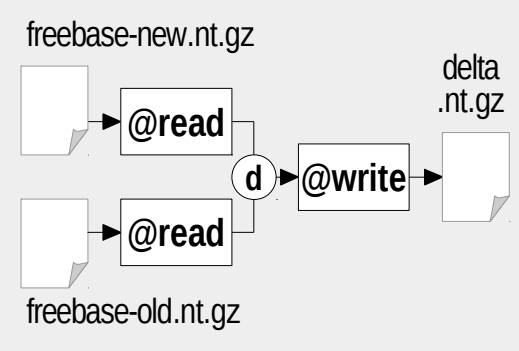
Experiment As a representative example of large-scale dataset analysis, we consider the tasks of extracting TBox and VOID statistics from Freebase data (2014/09/10 dump, 2863 MQ – millions of quads), whose schema and statistics are not available online, and

[94] Intel Core I7 860 CPU (4 cores, hyper-threading), 16 GB RAM, 500 GB 7200 RPM hard disk, Linux 2.6.32.

[95] `http://fracor.bitbucket.org/rdfpro/`



(a)



(b)

Task	Input		Output		Throughput		Time
	[MQ]	[MB]	[MQ]	[MB]	[MQ/s]	[MB/s]	[s]
1. TBox	2863	28339	0.23	3.01	1.43	14.12	2006
2. Statistics	2863	28339	0.13	1.36	0.34	3.36	8443
1-2. Aggregated	2863	28339	0.36	4.35	0.34	3.36	8426
3. Comparison	5486	55093	260	1894	0.42	4.25	12955

(c)

Figure 22: Dataset analysis flows (a, b) & results (c).

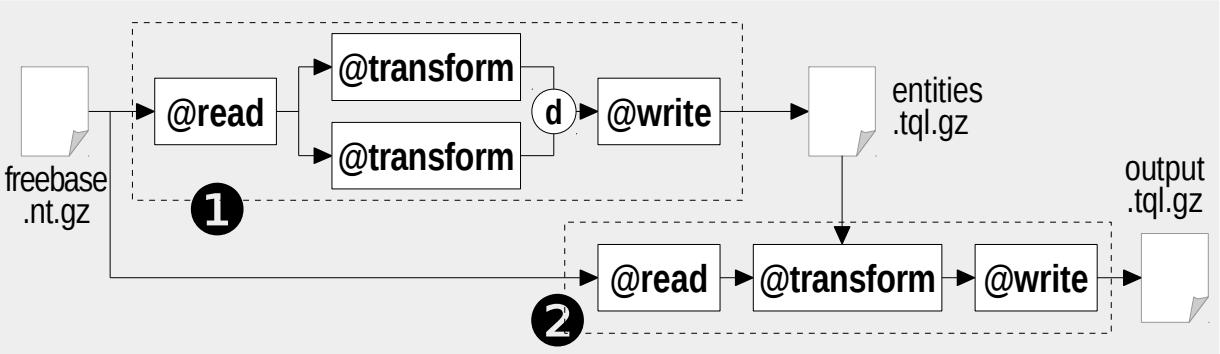
the task of comparing this Freebase release with a previous release (2014/07/10 dump, 2623 MQ) in order to identify newly added triples.[96]

We use the @tbox and @stats processors to extract TBox and VOID statistics, invoked both separately and aggregated in a pipeline processor as shown in Figure 22a. To extract new triples, we read both dataset releases and use parallel composition with the *difference* set operator (Section 6.1) to combine quads, as shown in Figure 22b.

Table 22c reports the tasks execution times, throughputs, input and output sizes both in quads and compressed (gzip) bytes as measured on our test machine. Additionally, when running the comparison task we measured a disk usage of 92.8 GB for the temporary files produced by the sorting-based *difference* set operator (∼18 bytes per input triple).

Comment Comparing the two Freebase releases resulted the most expensive task due to sorting and involved input size. When performed jointly, TBox and statistics extraction present performance figures close to statistics extraction alone, as data parsing is performed once and the cost of TBox extraction (excluded parsing) is negligible. This is an example of how the aggregation of multiple processing tasks in a single computation, enabled by RDF$_{\text{PRO}}$ streaming model and composition facilities, can generally lead to better performance due to a reduction of I/O overheads.

[96]From this delta TBox and VOID statistics can be extracted to get a concise summary of what has been added. This analysis is analogous to (and computationally cheaper than) the one done on the whole Freebase and is thus omitted.



(a)

Task	Input		Output		Throughput		Time
	[MQ]	[MB]	[MQ]	[MB]	[MQ/s]	[MB/s]	[s]
1. Select entities	2863	28339	0.20	0.73	1.36	13.4	2111
2. Extract quads	2863	28339	0.42	5.17	1.15	11.4	2481

(b)

Figure 23: Dataset filtering flow (a) and results (b).

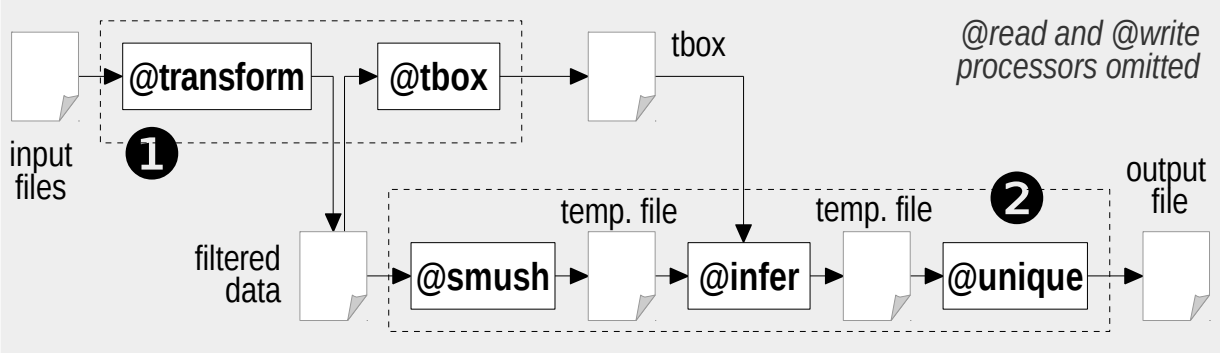
6.4.2 Dataset Filtering

When dealing with large RDF datasets, *dataset filtering* (or *slicing* [Marx *et al.*, 2013]) is often required to extract a small subset of interesting data, identified, e.g., based on a previous dataset analysis (Section 6.4.1). Dataset filtering typically consists in (i) identifying the entities of interest in the dataset, based on selection conditions on their URIs, rdf:type or other properties; and (ii) extracting all the quads about these entities expressing selected RDF properties. These two operations can be implemented using multiple streaming passes.

Experiment We consider a concrete dataset filtering example where the dataset is Freebase (2014/09/10 dump, 2863 MQ – millions of quads), the entities of interest are musical groups (i.e., their rdf:type is fb:music.musical_group) that are still active (i.e., there is no associated property fb:music.artist.active_end), and the properties to extract are the group name, genre and place of origin (respectively, rdfs:label, fb:music.artist.genre and fb:music.artist.origin).

We implement the task with two invocations of RDF$_{\text{PRO}}$ as in Figure 23a. The first invocation (marked as 1) generates an RDF file listing as subjects the URIs of the entities of interest; this is done with two parallel @transform processors, extracting respectively musical groups and no more active musical entities, whose outputs are combined using the *difference* set operator. The second invocation (marked as 2) uses another @transform processor to extract desired quads, testing predicates and requiring subjects to be contained in the previously extracted file (whose URIs are indexed in memory by a specific function in the @transform expression).

Table 23b reports the execution times, throughputs, input and output sizes of the two



(a)

Step	Input		Output		Throughput		Time
	[MQ]	[MB]	[MQ]	[MB]	[MQ/s]	[MB/s]	[s]
1. Transform	3394	33524	3394	36903	0.42	4.12	8137
2. TBox extraction	3394	36903	<1	4	1.28	13.9	2656
3. Smushing	3394	36903	3424	38823	0.37	3.98	9265
4. Inference	3424	38823	5615	51927	0.32	3.66	10612
5. Deduplication	5615	51927	4085	31297	0.33	3.03	17133
1-2. Aggregated	3394	33524	3394	36903	0.41	4.06	8247
3-5. Aggregated	3394	36903	4085	31446	0.14	1.56	23734

(b)

Figure 24: Dataset merging flow (a) and results (b).

invocations on the test machine.

Comment Although simple, the experiment shows how practical, large-scale filtering tasks are feasible using the streaming and sorting approach of RDF_{PRO}. Performance is worse than the ones obtainable using SPARQL in a triple store, but competitive if one considers also the time needed for indexing data in the triple store (see Section 6.4.5).

More complex filtering scenarios can be addressed using set operations for implementing conjunction, disjunction and negation of selection conditions, and with additional invocations of RDF_{PRO} that progressively augment the result (e.g., a third invocation can identify albums of selected artists, while a fourth invocation can extract the quads describing them). In cases where RDF_{PRO} model is insufficient (e.g., due to the need for aggregations or joins), the tool can still be used to perform a first coarse-grained filtering that reduces the number of quads and eases their downstream processing.

6.4.3 Dataset Merging

A common usage scenario is *dataset merging*, where multiple RDF datasets are integrated and prepared for application consumption. Data preparation typically comprises smushing, inference materialization and data deduplication (possibly with provenance tracking).

These tasks make the use of the resulting dataset more easy and efficient, as reasoning and entity aliasing have been already accounted for.

Experiment We consider a concrete dataset merging scenario with data from Freebase (2014/09/10 dump, 2863 MQ – millions quads), GeoNames (2013/08/27 dump, 125 MQ) and DBpedia in the four languages EN, ES, IT and NL (version 3.9, 406 MQ without redirects, disambiguation, pages and revisions metadata), for a total of 3394 MQ.

Figure 24a shows the required processing steps. A preliminary processing phase (marked as 1) is required to transform input data and extract the TBox axioms required for inference. Data transformation serves (i) to track provenance, by placing quads in different named graphs based on the source dataset; and (ii) to adopt optimal serialization format (Turtle Quads) and compression scheme (gzip) that speed up further processing. The main processing phase (marked as 2) consists in the cascaded execution of smushing, RDFS inference and deduplication to produce the merged dataset. Smushing identifies owl:sameAs equivalence classes and assigns a canonical URI to each of them. RDFS inference excludes rules *rdfs4a*, *rdfs4b* and *rdfs8* to avoid materializing uninformative ⟨ X rdf:type rdfs:Resource ⟩ quads. Deduplication takes quads with the same subject, predicate and object (possibly produced by previous steps) and merges them in a single quad inside a graph linked to all the original sources.

Table 24b reports the execution times, throughputs and input and output sizes of each step, covering both the cases where steps are performed separately via intermediate files and multiple invocations of RDF$_{\text{PRO}}$ (upper part of the table), or aggregated per processing phase using composition capabilities (lower part). Additionally, RDF$_{\text{PRO}}$ reported the use of ∼2 GB of memory for smushing an owl:sameAs graph of ∼38M URIs and ∼8M equivalence classes (∼56 bytes/URI).

Comment Also in this scenario, the aggregation of multiple processing tasks leads to a marked reduction of the total processing time (33% reduction from 47,803 s to 31,981 s) due to the elimination of the I/O overhead for intermediate files.

While addressed separately, the three scenarios of dataset analysis, filtering and merging are often combined in practice, e.g., to remove unwanted ABox and TBox quads from input data, merge remaining quads and analyze the result producing statistics that describe and document it; an example of such combination is reported on RDF$_{\text{PRO}}$ website.[97]

6.4.4 Dataset Massaging

We categorize three relevant, broad classes of *dataset massaging* tasks that are supported by RDF$_{\text{PRO}}$ processing model: data repackaging, data sanitization and data derivation.

Data repackaging comprises all the modifications that preserve data contents, i.e., the quads, and just affect the way data is packaged, i.e., the choices of RDF syntax, compression scheme and number of files. These modifications are often necessary to comply with restrictions of existing tools and systems, or to distribute data in a form that is best suited to the intended use (e.g., machine vs human consumption). Data repackaging operations

[97] `http://fracor.bitbucket.org/rdfpro/`

are all supported by RDF$_{PRO}$ and are best performed in a streaming model, which thus represent the most common choice for this task.

Data sanitization consists in fixing or removing the RDF terms or quads that prevent further processing of data. An example consists in the conversion of literals of a datatype to literals of an alternative datatype, because the former is not (properly) supported by the target system.[98] Other tasks supported in a streaming model include the rewriting of URIs (e.g., to change namespace), the normalization of literals (e.g., to ensure that rdfs:label strings obey certain capitalization patterns) and the removal of quads whose literal object has an excessive length.[99] These and similar tasks are supported by the RDF$_{PRO}$ @transform processor.

Data derivation consists in augmenting a dataset with quads computed from original data. Two broad classes of derivations supported in a streaming model are (i) quad-level derivations, and (ii) aggregations of quad-level information with emission of aggregate results at the end. Examples of the first kind include the conversion of a numeric value from a unit of measurement to another, as done in DBpedia "Mapping-based Properties (Specific)" dataset, or the computation of the age of persons starting from their birth dates. Examples of the second kind include counting the occurrences of a certain property for an entity (e.g., the number of person he/she foaf:knows). All these derivations are supported by @transform, while more complex derivations (e.g., involving joins) may in principle be implemented in new processors by exploiting the sorting primitive.

While we do not conduct experiments here, we note that the tasks described can be all implemented in a single pass without sorting. Assuming similar input and output sizes, performance roughly amounts to that of of reading and writing data in a pass (∼0.4-0.5 MQ/s on the test machine).

6.4.5 Discussion

The experimental results and the applicability of RDF$_{PRO}$ in relevant scenarios allow us to answer our research question and provide interesting findings on the use of our approach.

Research question assessment Two results emerge from the experiments: (i) RDF$_{PRO}$ implementation of the processing tasks of Section 1 succeeds in managing billions of RDF triples on a commodity machine; and (ii) execution times are in the order of hours (1h 16' for filtering 2.86 BQ, 5h 56' for analyzing 5.49 BQ, 8h 53' for merging 3.39 BQ).

The first result, which is trivial for tasks inherently expressible at a quad-level such as TBox extraction and some kinds of filtering, is not obvious for other tasks such as RDFS inference, smushing, statistics extraction, deduplication and set operations, for which we provide specialized implementations based on a mix of streaming and sorting techniques.

The second result can be put into perspective by comparing it with the time needed to load data in a triple store. On Virtuoso 7, a state-of-the-art triple store, the load time for

[98] E.g., the Community Edition of Virtuoso 7.1 normalizes xsd:gDay, xsd:gMonth and xsd:gYear values to xsd:datetime altering their semantics; changing to xsd:int is a partial fix.

[99] E.g., the OWLIM Lite 5.4.6486 triple store cannot store very long literals (e.g., 20M chars of GML geographic data).

one billion of quads is 9h 08' on our test machine and 27' on the very powerful machine used in the latest Berlin SPARQL Benchmark (BSBM) experiment.[100] Assuming that these rates hold for larger amounts of data, the comparison between these times and our processing times leads to two conclusions. First, given the same hardware, any *one-time* processing based on the use of a triple store—a common approach to RDF processing—is not competitive with our approach, as just the loading of input data would take longer than our processing time in the considered scenarios.[101] Second, our processing times are negligible if compared to load times on the same machine, and have the same order of magnitude of load times in the BSBM machine, overall meaning that RDF processing based on RDF_{PRO} approach would not slow down (and is thus compatible with) a typical *Extract, Transform, Load* (ETL) procedure where resulting RDF data is put in a production triple store.

Based on these results, a positive answer can be given to the research question *"Are relevant RDF processing tasks practically feasible on large datasets by using streaming and sorting techniques on a single commodity machine?"*.

Other findings The empirical evaluation also highlights the importance of task aggregation and allows us to analyze the factors impacting streaming and sorting performance.

Aggregation of multiple processing tasks in a single RDF_{PRO} invocation provides better performance, because input data is parsed once and I/O costs for accessing intermediate files are eliminated, as shown in Section 6.4.1 and 6.4.3. Task aggregation requires composition primitives and the support for reading and writing data at any point of the pipeline, two features of RDF_{PRO} that are relevant to any similar tool.

Streaming performance within a pipeline are highly dependent on the file compression and RDF format used. No compression and best compression (e.g., `bzip2` used in DBpedia dumps) are inefficient, with `gzip` representing a good trade-off; using native compression utilities (and especially their parallel versions, e.g., `pigz` and `pbzip2`) is also beneficial. Line-oriented RDF formats such as NTriples and NQuads provide better performance as they allow for multi-thread parsing and serialization (e.g., from 0.61 MQ/s to 1.45 MQ/s for Freebase NTriples+gzip data with multiple threads). We also experimented with the HDT binary format [Fernández *et al.*, 2013], but writing HDT is very expensive while reading HDT is not faster than reading other formats (unless lookup of RDF terms in the HDT dictionary is skipped).

Sorting performance depends on a number of factors. In our experience, performance can be improved by allocating a large amount of memory for sorting (we gave 8 GB out of the available 16 GB to the `sort` utility in our experiments), by using a parallel sort implementation and by configuring the compression of temporary files. Dictionary encoding of frequent RDF terms also helps to improve throughput.

[100] 2x Intel Xeon E5-2650 CPU (8 cores, hyper-threading), 256 GB RAM, 3x 1.8 TB 7200 RPM disks RAID 0, Linux 3.3.4; see `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V7/index.html`

[101] Of course, using a triple store may pay off in scenarios where data is loaded once and processed many times, or when the triple store is also the final destination of data.

7 The ESO reasoner

Changes wrt the KnowledgeStore Deliverable D6.2.1

- Brand new section.

One of the NewsReader objectives is the representation of events and their effects on the entities involved in them. For example, in the "giving" event, there would be two people and an object involved: a person gives something to another person at time T. The event also marks a change to the entities: at the time T, person A owns the object, while person B does not (aka, *pre-situation*); after T, the situation is inverted (aka, *post-situation*).

To describe these facts, the Event and Situation Ontology (ESO) has been developed. [Segers *et al.*, 2015] It defines two main classes of entities: *events* and *situations*. An *event* describes a change in the world (for instance, person A gives an object to person B at time T); typically an event has a certain number of participants (the two people and the object) and a *time* (in the example, T) associated to it. A *situation* is an entity that associates a period of time to a set of statements. In the example ("person A gives an object to person B at time T"), we can identify a *pre-situation* (before time T):

- person A owns the object
- person B does not own the object

and a *post-situation* (after time T):

- person A does not own the object
- person B owns the object.

For instance, in the sentence

> Lincoln City sacked manager Allan Clarke on November 30, 1990 and appointed Steve Thompson in his place.

two events are represented: (i) Lincoln City fires Allan Clarke; (ii) Lincoln City hires Steve Thompson. We will describe how the reasoner works using the first event as an example.

Event (i) is represented as follows:

- *⟨#evID, rdf:type, sem:Event⟩*
- *⟨#evID, rdf:type, eso:LeavingAnOrganization⟩*
- *⟨#evID, rdfs:label, sack⟩*
- *⟨#evID, eso:employment-employer, dbp:Lincoln_City_F.C.⟩*
- *⟨#evID, eso:employment-employee, dbp:Allan_Clarke_(footballer)⟩*
- *⟨#evID, sem:hasTime, #tmxID⟩*

- ...

where *#tmxID* is the RDF representation of "November 30, 1990".

In the ESO, the event *eso:LeavingAnOrganization* has both *pre-* and *post-situation*, *eso:employedAt* and *eso:notEmployedAt*, respectively.

The reasoner would add the two situations to the triple store, adding links to the event to which they belong. In the *pre-situation* of our example, Allan Clarke works for Lincoln City, while in the *post-situation* he does not work for the company any more. Therefore, the reasoner generates the following set of quads:

- *⟨#evID, eso:hasPostSituation, #evID_post⟩*
- *⟨#evID_post, rdf:type, eso:Situation⟩*
- *⟨#evID_post, sem:hasTime, #tmxID⟩*
- *⟨dbpedia:Allan_Clarke_(fb), eso:notEmployedAt, dbp:Lincoln_City_F.C., #evID_post⟩*
- *⟨#evID, eso:hasPreSituation, #evID_pre⟩*
- *⟨#evID_pre, rdf:type, eso:Situation⟩*
- *⟨#evID_pre, sem:hasTime, #tmxID⟩*
- *⟨dbpedia:Allan_Clarke_(fb), eso:employedAt, dbp:Lincoln_City_F.C., #evID_pre⟩*

Section 7.1 shows how the ESO reasoner works and describes its implementation. It can be downloaded from the GitHub repository.[102] An exhaustive description of classes and properties is available in [Segers *et al.*, 2015].

7.1 The tool

The tool is developed as a dedicated processor (@esoreasoner) of RDF$_{\text{PRO}}$ (see Section 6), and it combines OWL DL reasoning and a simple rule engine. It is released as an open source package and can be downloaded from its BitBucket repository.[103]

For any event identified in the text (see [Rospocher *et al.*, 2014c]), the module applies OWL reasoning to identify the ESO trigger rules (if any) to be applied on that event, and based on the roles attached to the event, it instantiates the corresponding implications according to the rules. As the ESO reasoner reads the rules it applies directly from the ESO Domain Ontology (i.e., rules are not hard-coded in the module), rules can be revised or adapted without any adaptation on the module itself.

The workflow of the tool is as follows:

- Using the RDF$_{\text{PRO}}$ streaming and sorting paradigm, the input is filtered: only RDF triples concerning events are passed to the reasoner. In addition, data is sorted, therefore the triples are grouped by event URI (for instance, *E*).

[102] `https://github.com/newsreader/eso`
[103] `https://github.com/dkmfbk/eso-reasoner`

- The event types associated to E are retrieved (for example, let E be a *eso:Leaving-AnOrganization* event).
- The ontology is queried to know which attributes should be looked at, and their values for E are retrieved (this corresponds to *eso:employment-employee*, *eso:employment-employer*, and *sem:Time* in the example).
- For each triggersPreSituation/triggersPostSituation assertion (i.e. situation) attached to the event type (in the example, this correspond to *eso:LeavingAnOrganization_Employment_before* and *eso:LeavingAnOrganization_Employment_after* in our ontology; let's consider just the *eso:LeavingAnOrganization_Employment_before*) the following steps are carried out:
  - A new named graph of type Situation is created, instantiating the time aspect according to the *eso:hasTimeScope* assertion (a new named graph *Situation_pre_1_E* is created, it is a *eso:Situation*; since the object of *eso:hasTimeScope* is *time:hasEnd*, an assertion *⟨Situation_pre_1_E, time:hasEnd, time12345⟩* is instantiated)
  - For each *eso:hasAssertion* attached to a situation (i.e., *eso:SituationAssertion*), do:
    - Check if the object of *eso:hasSubject*, *eso:hasObject* assertions are actually stated for event E: let them be triples t_1 and t_2, that may actually be sets of triples (in the example, the objects of *eso:hasSubject* and *eso:hasObject* are *eso:employment-employee* and *eso:employment-employer*; let E have assertion for both of them).
    - Let P be the object of the *eso:hasProperty* assertion of the current *eso:SituationAssertion* (for instance, it could be *eso:isEmployedAt*).
    - A new triple *⟨a, p, b⟩* is added for each $(a, b) \in t_1 \times t_2$.

The reasoner has been tested in the Global Automotive Industry dataset (Scenario 3, see Section 5.4.3).

8 Conclusions and Future Work

In this deliverable we documented the current implementation 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, outlining the selection process and the processing pipeline to extract, combine and augment relevant data to produce a coherent dataset.

This version of the NewsReader KnowledgeStore provides the core infrastructure functionalities and several improvements have been made compared to the previous release: the implementation of the user interface (Section 3.3); some updates on the population (including three different case studies, showing the scalability of the KnowledgeStore, see Section 5); the description of some new features in the KnowledgeStore architecture (Section 4); the development of a tool (RDF_{PRO}, see Section 6); the new ESO reasoner (Section 7).

Several research challenges worth investigating are identified for the future. We will further assess the scalability performance (in populating, querying, retrieving) of the KnowledgeStore, using larger datasets than the ones considered so far. Concerning the reasoning services, we will investigate the possibility of extending the number (and type of) triples inferred from extracted events by exploiting the ESO rules and ontology: for instance, by inferring from a "leaving an organization" event that *entityA* is not working anymore for *entityB*, we can derive additional triples stating that *entityA* is a person and *entityB* an organization, and these assertions can be checked against known types of *entityA* and *entityB* for consistency.

Building on top of the reasoning module, we also plan to develop some techniques to clean/crystallize the triples extracted from text, in order to increase the quality of the content injected into the KnowledgeStore (e.g., by filtering (or weakening) assertions that are not logically consistent either with previously extracted content or with the content inferred via the ESO rules).

Concerning the retrieval of content stored in the KnowledgeStore, we plan to investigate the possibility of offering additional query mechanisms, based on indexing technologies such as Apache Lucene and Solr, enabling support for (i) full text search on the documents stored into the KnowledgeStore, and (ii) *mixed* queries, i.e., queries that either in the formulation/expansion or in the result set combine both structured and unstructured aspects (e.g., answering a query both by accessing the triple store component of the KnowledgeStore and the full text index built on its document part).

Finally, to improve the user experience in exploring and navigating the KnowledgeStore content, we will investigate the possibility of integrating techniques based on faceted searching[104] and browsing[105] on top of the KnowledgeStore.

[104] http://en.wikipedia.org/wiki/Faceted_search
[105] http://www.sindicetech.com/pivotbrowser.html

References

[Aggarwal *et al.*, 2004] Gagan Aggarwal, Mayur Datar, Sridhar Rajagopalan, and Matthias Ruhl. On the streaming model augmented with a sorting primitive. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 540–549, 2004.

[Alexander *et al.*, 2009] Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. Describing linked datasets. In *Workshop on Linked Data on the Web (LDOW)*, 2009.

[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.

[Ceri *et al.*, 1989] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Knowl. Data Eng.*, 1(1):146–166, 1989.

[Chandrasekar *et al.*, 2013] S. Chandrasekar, R. Dakshinamurthy, P.G. Seshakumar, B. Prabavathy, and C. Babu. A novel indexing scheme for efficient handling of small files in hadoop distributed file system. In *Computer Communication and Informatics (ICCCI), 2013 International Conference on*, pages 1–8, Jan 2013.

[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).

[Corcoglioniti *et al.*, 2014a] Francesco Corcoglioniti, Marco Rospocher, Marco Amadori, and Michele Mostarda. RDFpro: an extensible tool for building stream-oriented RDF processing pipelines. In *Proc. of ISWC Developers Workshop colocated with 13th Int. Semantic Web Conference (ISWC'14), Riva del Garda, Italy*, volume 1268 of *CEUR Workshop Proceedings*, pages 49–64. CEUR-WS, 2014.

[Corcoglioniti *et al.*, 2014b] Francesco Corcoglioniti, Marco Rospocher, Marco Amadori, and Michele Mostarda. RDFpro: an extensible tool for building stream-oriented RDF processing pipelines. In *Proc. of ISWC Developers Workshop colocated with 13th Int. Semantic Web Conference (ISWC'14), Riva del Garda, Italy*, CEUR Workshop Proceedings. CEUR-WS, 2014. to appear.

[Corcoglioniti *et al.*, 2015] Francesco Corcoglioniti, Marco Rospocher, Michele Mostarda, and Marco Amadori. Processing billions of rdf triples on a single machine using streaming and sorting. In *ACM SAC 2015 Proceedings*, 2015.

[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.

[Dong *et al.*, 2010] Bo Dong, Jie Qiu, Qinghua Zheng, Xiao Zhong, Jingwei Li, and Ying Li. A novel approach to improving the efficiency of storing and accessing small files on hadoop: A case study by powerpoint files. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 65–72, July 2010.

[Erxleben *et al.*, 2014] Fredo Erxleben, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandečić. Introducing Wikidata to the Linked Data Web. In

Proc of 13th Int. Semantic Web Conference (ISWC'14), volume 8796 of *Lecture Notes in Computer Science*, pages 50–65. Springer International Publishing, 2014. http://dx.doi.org/10.1007/978-3-319-11964-9_4.

[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/.

[Fernández *et al.*, 2013] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary RDF representation for publication and exchange (HDT). *Web Semant.*, 19:22–41, 2013.

[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.

[Fokkens *et al.*, 2013] Antske Fokkens, Marieke van Erp, Piek Vossen, Sara Tonelli, Willem Robert van Hage, Luciano Serafini, Rachele Sprugnoli, and Jesper Hoeksema. GAF: A grounded annotation framework for events. In *Proceedings of the first Workshop on Events: Definition, Dectection, Coreference and Representation*, Atlanta, USA, 2013.

[Fokkens *et al.*, 2014] Antske Fokkens, Aitor Soroa, Zuhaitz Beloki, Niels Ockeloen, German Rigau, Willem Robert van Hage, and Piek Vossen. NAF and GAF: Linking linguistic annotations. In *To appear in Proceedings of 10th Joint ACL/ISO Workshop on Interoperable Semantic Annotation (ISA-10)*, 2014.

[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.

[Hopkinson *et al.*, 2014] Ian Hopkinson, Steve Maude, and Marco Rospocher. A simple api to the knowledgestore. In *Proc. of ISWC Developers Workshop colocated with 13th Int. Semantic Web Conference (ISWC'14), Riva del Garda, Italy*, CEUR Workshop Proceedings. CEUR-WS, 2014. to appear.

[Lehmann *et al.*, 2014] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 2014.

[Marx *et al.*, 2013] E. Marx, S. Shekarpour, S. Auer, and A-C.N. Ngomo. Large-scale RDF dataset slicing. In *IEEE Int. Conf. on Semantic Computing (ICSC)*, pages 228–235, 2013.

[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/`.

[O'Connell, 2009] ThomasC. O'Connell. A survey of graph algorithms under extended streaming models of computation. In *Fundamental Problems in Computing*, pages 455–476. Springer Netherlands, 2009.

[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`.

[Rospocher *et al.*, 2014a] Marco Rospocher, Francesco Corcoglioniti, Roldano Cattoni, Bernardo Magnini, and Luciano Serafini. Integrating nlp and sw with the knowledgestore. In *ISWC 2014 Posters and Demonstrations Track, within the 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Italy, October 21, 2014*, volume 1272 of *CEUR Workshop Proceedings*, pages 69–72. CEUR-WS.org, 2014.

[Rospocher *et al.*, 2014b] Marco Rospocher, Francesco Corcoglioniti, Roldano Cattoni, Bernardo Magnini, and Luciano Serafini. Integrating unstructured and structured knowledge with the knowledgestore. In *Proceedings of the Posters and Demos of the 19th International Conference on Knowledge Engineering and Knowledge Management (EKAW2014)*, 2014.

[Rospocher *et al.*, 2014c] Marco Rospocher, Piek Vossen, Tommaso Castelli, and Agata Cybulska. Deliverable d5.1.2, 2014.

[Segers *et al.*, 2015] Roxane Segers, Piek Vossen, Marco Rospocher, Luciano Serafini, Egoitz Laparra, and German Rigau. Eso: A frame based ontology for events and implied situations. In *Proceedings of Maplex2015*, 2015.

[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 and Ploeger, 2014] Willem Robert van Hage and Thomas Ploeger. Deliverable d7.3.1, 2014.

[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.

[Welsh *et al.*, 2001] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243, 2001.