KnowledgeStore version 3
Deliverable D6.2.3
Version FINAL

Authors: Francesco Corcoglioniti¹, Marco Rospocher¹, Roldano Cattoni¹, Marco Amadori¹, Bernardo Magnini¹, Mohammed Qwaider¹, Michele Mostarda¹, Alessio Palmero Aprosio³, Luciano Serafini¹, Loris Bozzato¹

Affiliation: (1) FBK

Building structured event indexes of large volumes of financial and economic data for decision making
ICT 316404
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 various versions of the KnowledgeStore were populated is also provided.
# Table of Revisions

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description and reason</th>
<th>By</th>
<th>Affected sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>18 December 2013</td>
<td>Starting new draft from D6.2.1</td>
<td>Francesco Corcogli...</td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>20 July 2014</td>
<td>Added user interface</td>
<td>Francesco Corcogli...</td>
<td>3.4</td>
</tr>
<tr>
<td>0.3</td>
<td>15 August 2014</td>
<td>Added backends</td>
<td>Alessio Palmero Aprosio</td>
<td>4.1.6</td>
</tr>
<tr>
<td>0.3.1</td>
<td>30 November 2014</td>
<td>Updated NAF populator</td>
<td>Roldano Cattoni</td>
<td>5</td>
</tr>
<tr>
<td>0.4</td>
<td>15 December 2014</td>
<td>Added RDFpro</td>
<td>Francesco Corcogli...</td>
<td>7</td>
</tr>
<tr>
<td>0.5</td>
<td>20 December 2014</td>
<td>Added ESO reasoner</td>
<td>Alessio Palmero Aprosio</td>
<td>8</td>
</tr>
<tr>
<td>1.0</td>
<td>10 January 2015</td>
<td>Revision of whole document</td>
<td>Alessio Palmero Aprosio</td>
<td></td>
</tr>
<tr>
<td>1.0.1</td>
<td>14 January 2015</td>
<td>Updated background knowledge</td>
<td>Francesco Corcogli...</td>
<td>5.3</td>
</tr>
<tr>
<td>1.1</td>
<td>20 January 2015</td>
<td>Added use cases</td>
<td>Francesco Corcogli...</td>
<td>5.4</td>
</tr>
<tr>
<td>1.1</td>
<td>25 January 2015</td>
<td>Review minor editorial changes</td>
<td>Antske Fokkens</td>
<td>All</td>
</tr>
<tr>
<td>1.1</td>
<td>30 January 2015</td>
<td>Check by coordinator</td>
<td>VUA</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>20-25 October 2015</td>
<td>Revision of whole document, added new sections (e.g., Related Work)</td>
<td>Alessio Palmero Aprosio</td>
<td>All</td>
</tr>
<tr>
<td>1.3</td>
<td>26 October 2015</td>
<td>Revision of whole document</td>
<td>Marco Rospocher</td>
<td>All</td>
</tr>
</tbody>
</table>
Executive Summary

This deliverable documents the final 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), Task 6.2 (KnowledgeStore implementation and filling) and Task 6.3 (KnowledgeStore reasoning services) 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 the 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 unstructured and structured content from various domains (e.g., Global Automotive Industry, General Domain News), spanning from few thousands to millions of news articles, as well as selected structured resources coming from DBpedia.org (2015 Version), the core repository of the Linked Data cloud. We also present the reasoning services developed as part of rdfpro, a public domain, Java command line tool and library for RDF processing.

To assess the KnowledgeStore scalability, we report on a thorough evaluation of its performances, covering both data population and data retrieval with different dataset sizes and numbers of concurrent clients.

Before concluding, we position the KnowledgeStore with respect to other state of the art contributions for the integrated and interlinked storage of unstructured and structured content.

Part of the contributions here described was presented at the 7th IEEE International Conference on Semantic Computing (ICSC2013) [Corcoglioniti et al., 2013], the ISWC 2014 Posters and Demonstrations Track [Rospocher et al., 2014a], the ISWC 2014 Developers Workshop [Corcoglioniti et al., 2014a], the Posters and Demos track of the 19th International Conference on Knowledge Engineering and Knowledge Management (EKAW2014) [Rospocher et al., 2014b], ACM SAC 2015 [Corcoglioniti et al., 2015c], the ISWC 2015 Posters and Demonstrations Track [Corcoglioniti et al., 2015a], the DeRiVE Workshop at ESWC2015 [Bozzato et al., 2015a], and the CILC2015 Conference [Bozzato et al., 2015b].
The KnowledgeStore has also been described in the International Journal on Semantic Web and Information Systems [Corcoglioniti et al., 2015b].
# Contents

**Table of Revisions**

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

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

3. **The KnowledgeStore Interfaces**
   - 3.1 API Design Criteria
   - 3.2 API Operations and Endpoints
     - 3.2.1 CRUD Endpoint
     - 3.2.2 SPARQL Endpoint
   - 3.3 The custom API
     - 3.3.1 Streaming population of the KnowledgeStore
   - 3.4 The KnowledgeStore User Interface
     - 3.4.1 Reports and filters

4. **The KnowledgeStore Architecture and Implementation**
   - 4.1 Architecture
     - 4.1.1 HBase & Hadoop
     - 4.1.2 ElasticSearch
     - 4.1.3 The MultiFileStore
     - 4.1.4 Virtuoso
     - 4.1.5 Frontend Server
   - 4.2 Implementation
     - 4.2.1 Software development
     - 4.2.2 Deployment environments
   - 4.3 Distribution

5. **The KnowledgeStore Population**
   - 5.1 NAF populator
     - 5.1.1 Multi-Thread
   - 5.2 RDF populator
   - 5.3 Acquisition of LOD background knowledge
     - 5.3.1 Data selection
     - 5.3.2 Data processing
     - 5.3.3 Results
   - 5.4 The KnowledgeStore in action: use cases
**List of Tables**

1. Disk usage of ElasticSearch on the WikiNews corpus ................................................. 42
2. Performance of the MultiFileStore .................................................................................. 44
3. Data selection criteria ....................................................................................................... 58
4. LOD datasets candidate for inclusion in the background knowledge. .......................... 59
5. Number of triples in produced datasets. ........................................................................... 64
6. Number of entities in produced datasets. ....................................................................... 64
7. Processing statistics. ......................................................................................................... 65
8. The mostFrequentFrameElements query on the KnowledgeStore. ............................. 75
9. Population time and rate depending on number of mentions per news article. ............ 78
10. Population rate depending on amount of data already stored. We recall that Resources include both the original news articles as well as its NAF annotated version (i.e., the number of resources is twice the number of news articles). .............................. 78
11. The test datasets (from smaller to larger, in terms of number of resources). ................ 80
12. Parametric requests used in the test. .............................................................................. 80

**List of Figures**

1. KnowledgeStore Content .................................................................................................. 12
2. The role of the KnowledgeStore in NewsReader. .............................................................. 15
3. KnowledgeStore data model. ............................................................................................ 18
4. From RDF statements to axioms. ....................................................................................... 19
5. Representation of axioms with context and metadata using named graphs. .................. 20
6. Example of axiom representation using named graphs. .................................................. 20
7. NewsReader data model. ................................................................................................... 21
8. Invocation of CRUD retrieve operation through the HTTP ReST endpoint. .................. 30
9. Using the KnowledgeStore client within a Java application. ......................................... 30
10. SPARQL endpoint example. ............................................................................................ 31
11. The KnowledgeStore UI .................................................................................................. 35
12. Using reports to search the dbpedia:Fiat entity. ............................................................... 37
13. KnowledgeStore architecture. ......................................................................................... 39
14. Axiom representation in HBase and in the Virtuoso Triple Store ..................................... 43
15. Examples of inference rules. ............................................................................................. 45
16. Examples of generated reports on the KnowledgeStore web site.................................. 48
17. Modular code organization. ............................................................................................... 49
18. NAF population. ............................................................................................................... 53
19. NAF Multi-Threading populator. ....................................................................................... 55
20. Example of SPARQL query with (a) and without (b) smushing and inference. ........... 62
21. Examples of browsing the statistics ontology in Protégé. ............................................... 66
Example of output obtained running Linking Analyzer against a populated KnowledgeStore instance.

Request throughput (a) and average evaluation time (b) with different clients and dataset sizes.

Average evaluation time of each parametric request using one client and varying dataset sizes.

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

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

Dataset filtering flow (a) and results (b).

Dataset merging flow (a) and results (b).

Using the command line (a) and web (b) interfaces of RDFPRO.

CKR structure.

Example event in CKR-ESO model.
1 Introduction

This prototype deliverable presents the implementation of the third 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, and updated in Deliverable D6.2.2. This revision updates the KnowledgeStore design in light of the latest outcomes of some activities: the implementation of the user interface (Section 3.4); 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 (RDFPRO, see Section 7); the new ESO reasoner (Section 8). Finally, Section 9 compare this work with other existing approaches managing unstructured and structured content, to address the request of the project reviewers to highlight the differences between the KnowledgeStore and other state-of-the-art frameworks.

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://knowledgestore.fbk.eu;
- a KnowledgeStore publicly accessible installation, populated with unstructured and structured content (extracted) from Wikinews available at http://knowledgestore2.fbk.eu/nwr/wikinews;

Descriptions from Deliverables D6.2.1 and D6.2.2 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. These indications are meant to point readers familiar with the content of D6.2.1 and D6.2.2 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\footnote{http://linkeddata.org} initiative), 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 \cite{GantzReinsel2011}, 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 \cite{Ferrucci2010}, 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 an entity or an axiom of the entity layer. Mentions can be automatically extracted by natural language and multimedia processing tools, that can enrich them with additional attributes 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. However, we highly revised the design of the previous version, introducing significant enhancements: this 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 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

---

3 [http://www.livememories.org/](http://www.livememories.org/)

4 [http://www.newsreader-project.eu/](http://www.newsreader-project.eu/)
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;
- 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., DBPedia⁵ corporate databases);
- to interrelate entities (events and their participants, in particular) to support the construction of narrative stories;
- to reason over events to check consistency, completeness, factuality and relevance;
- to store all this huge quantity of information (on resources, mentions, entities) in a scalable way, enabling efficient retrieval and intelligent queries;
- to effectively offer narrative stories to decision makers.

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

First, the KnowledgeStore allows 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 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 Figure² 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:

⁵http://dbpedia.org/
⁶Note 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.
⁷In 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.
• 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 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.

• 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.

• 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.

---

8These documents represent text and linguistic annotations in the NLP Annotation Format ([Fokkens et al., 2014], NAF)
provided, such that external modules have full access (and control) on the content of the KnowledgeStore.

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.

\footnote{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 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

The data model defines what information can be stored in the KnowledgeStore, in accordance with the annotation guidelines of WP3, the event modeling activity of WP5 and the NLP Annotation Format (NAF) and Grounded Annotation Framework developed as part of WP2. 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 by reusing terms from external vocabularies and providing alignments to concepts in the Dolce+DNS Ultralite upper ontology. It includes:

<table>
<thead>
<tr>
<th>Notes:</th>
<th>Notes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The revised guidelines will be described in Deliverable D3.3.1: Annotated Data. The NLP Annotation Format (NAF) 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. It includes:</td>
<td>NAF is described in Deliverables D2.1 and D2.2: System Design. It includes:</td>
</tr>
</tbody>
</table>
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 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 statements 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 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:

NewsReader: ICT-316404   February 8, 2016
• 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\(^\text{14}\) 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.

\(^{14}\)As of 2013/12/15. Minor changes may occur to best accommodate the NAF output of WP4 pipeline.
Figure 7: NewsReader data model.
of WP3 (see Deliverable D3.3.2: Annotated Data) as well as the latest NAF specification (see Deliverable D2.2: System Design). The OWL 2 ontology formally encoding the model is available online[15]. 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 NAF-Document 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[16] (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:
  - 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, geopolitical entities or locations, events, points and intervals in time extracted from text; the type of entity is conveyed by an rdf:type axiom. The context in which an axiom holds is described and identified in terms of temporal validity (sem:hasTimeValidity) and time-referenced point of view (sem:hasPointOfView, e.g., “Financial Times” point of view expressed on 2013/12/15); the Simple Event Model (SEM) [van Hage et al., 2011] and the OWL Time 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).\(^{18}\) 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.

\(^{18}\)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

<table>
<thead>
<tr>
<th>Changes wrt the KnowledgeStore Interfaces described Deliverable D6.2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>• added the new custom API (Section 3.3);</td>
</tr>
<tr>
<td>• added reports and filters in the UI (Section 3.4);</td>
</tr>
</tbody>
</table>

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.

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

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.\(^{20}\) 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 \(^{21}\) rules out consistency, and thus ACID in a strict sense.\(^{21}\) 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.\(^{22}\) Violations of these constraints may either be reported as warnings...
or may cause the API request to fail.

**Security** Access to the KnowledgeStore API must be restricted only to authorized clients, since it allows 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 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 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. 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](https://newsreader.fbk.eu/knowledgestore) or in [Tao et al., 2010](https://newsreader.fbk.eu/knowledgestore) 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.

---

23 Accessible from the KnowledgeStore website: [https://newsreader.fbk.eu/knowledgestore](https://newsreader.fbk.eu/knowledgestore)

24 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.

25 Accessible from the KnowledgeStore website: [https://newsreader.fbk.eu/knowledgestore](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). 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.

---

26 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.²⁷

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.

²⁷With 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.
Figure 8: Invocation of CRUD retrieve operation through the HTTP ReST endpoint.

```java
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();
}
```

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:

<table>
<thead>
<tr>
<th>share</th>
<th>from</th>
<th>to</th>
<th>authority</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.6%</td>
<td>2007-01-01</td>
<td>2007-12-31</td>
<td>dbpedia:Forbes</td>
</tr>
<tr>
<td>12.3%</td>
<td>2011-01-01</td>
<td>2011-12-31</td>
<td>dbpedia:Forbes</td>
</tr>
</tbody>
</table>

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, 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:

- **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.

---

28[http://www.w3.org/wiki/SPARQL](http://www.w3.org/wiki/SPARQL)

29The definition of the `sparqlQuery()` operation is based on the SPARQL protocol standard [Feigenbaum et al., 2013](http://www.w3.org/wiki/SPARQL); 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 \( \langle \text{subject, predicate, object} \rangle \) 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 custom API

To enable a more flexible customization of the KnowledgeStore (e.g., to support more abstract content editing operations than CRUDs), a new API called custom has been developed. By using it, a user is able to interact live with data stored in the KnowledgeStore, reading and writing documents, document metadata, and triples. See below (Section 3.3.1) for a specific example of custom API defined to support the incremental population and coreference materialization of events extracted from text.

The implementation is based on RDFPRO (see Section 7), and the custom operation to be performed has to be defined as a combination of custom RDFPRO processors.

The user defines a new label (for example, readTriples) in the configuration file, specifying which RDFPRO command should be executed once the function is called. The POST data sent to the call is processed as the input of the RDFPRO pipeline (the @read processor is automatically used, without need to specify it). The Content-Type header is used to understand the input format (Turtle, TRiG, and so on).

Custom configurations can be added to the ks.ttl file, using the customConfigs directive. In the following example, a custom API called backup is used to write the provided data (using the @write processor) in the file /tmp/file.tql. The .tql extension tells RDFPRO the format to use.

```pro
:customConfigs [ 
  a <java:eu.fbk.knowledgestore.server.http.CustomConfig> ; 
  :name "backup" ; 
  :command "@write /tmp/file.tql"
] ;
```

NewsReader: ICT-316404	February 8, 2016
3.3.1 Streaming population of the KnowledgeStore

An example custom API instantiation is the invocation of the @naf2sem processor of rdFpro. This command is available as a part of the ESO reasoner source code\(^{30}\) (see Section\(^{8}\)); using it, documents and triples can be added incrementally to the KnowledgeStore, so that data can be populated on-the-fly in it.

For each bunch of new documents that must be populated into the KnowledgeStore, the procedure can be done in two steps:

- first, the NAF document is loaded using the NAF populator (see Section\(^{5.1}\)); in this phase, the resource is uploaded to the KnowledgeStore and the corresponding metadata and mentions are extracted and stored in the data store (HBase, ElasticSearch, or one of the alternative backends, see Section\(^{4.1}\));

- then, the @naf2sem custom API is called, see below.

The flow of the @naf2sem can be summarized as follows.

1. The input data to be populated is provided inside the POST data in the call. The input format can be specified using the Content-Type header in the request. Every format accepted by rdFpro can be used (see Section\(^{7}\)).

2. A backup of the input data is saved on the disk (if the backup folder is provided to the processor; otherwise, this step is ignored).

3. The data contains events extracted from the documents. This extraction is performed by the Event Narrative Module \cite{Vossen2015}. The process takes into account the events extracted from documents already processed and stored in the KnowledgeStore, therefore the input data can contain the connection between these events and the new ones freshly extracted from the new documents. This connection is performed using owl:sameAs between events. The @naf2sem processor can also deal with the possibility that the merge involves two or more events already stored in the KnowledgeStore: in this case, it is sufficient that the data contains the owl:sameAs between the corresponding events. Hence, the owl:sameAs relations between events are collected, and the list of statements related to that events are retrieved for the @smush phase (see Step 5).

4. Some pre-processing scripts (written in Groovy\(^{31}\) and included in the source code\(^{32}\)) are executed on the input data.

5. Events participating in a sameAs relation, already collected, are smushed\(^{33}\) in a single event.

\(^{30}\)https://github.com/dkmfbk/eso-reasoner
\(^{31}\)http://www.groovy-lang.org/
\(^{32}\)https://github.com/dkmfbk/eso-reasoner/tree/master/src/main/resources
\(^{33}\)http://www.w3.org/wiki/RdfSmushing
6. The ESO reasoner is applied to extract situations (see Section 8).

7. Some obsolete data (such as information related to smushed events) is removed from the triple store.

8. Finally, the resulting data (RDF quads) are submitted to the triple store.

From a technical point of view, this flow can be used by defining a new custom API (called `naf2sem`, for instance) in the `KnowledgeStore` configuration file, using the following syntax:

```
:customConfigs [
  a <java.eu.fbk.knowledgestore.server.http.CustomConfig> ;
  :name "naf2sem" ;
  :command "@naf2sem -b [backup-dir] -k [ks-address] -o [eso-ontology-file]"
]
```

### 3.4 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). 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 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`

---

[34] The features of the `KnowledgeStore` UI are comprehensively showcased in a demo video [http://youtu.be/if1PRwsS15c](http://youtu.be/if1PRwsS15c).
(a) Resource Lookup

(b) Mention Lookup

(c) Running a SPARQL query

Figure 11: The KnowledgeStore UI
(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.

### 3.4.1 Reports and filters

We added some new methods for accessing, within the same request, content stored in different layers of the KnowledgeStore (aka, *mixed queries*). In particular, four new methods are accessible from the KnowledgeStore User Interface, under the tab “Report” of any KnowledgeStore installation:

- **Entity mentions**: given the URI of an entity (e.g., `dbpedia:Barack_Obama`), it returns all the attributes associated to mentions corresponding to that entity. An additional *property* (and *property value*) parameter can be set, to obtain only those mentions having that property as attribute (and that property value as attribute value);

- **Entity mentions (aggregate)**: given the URI of an entity (e.g., `dbpedia:Barack_Obama`), for each mention attribute - attribute value pair, the number of mentions having that value for that attribute is returned, together with an example mention. By clicking on the number of mentions, all the mentions having that value for that attribute are returned, via an invocation of the previous method.

- **Mention value occurrences**: given the URI of an entity (e.g., `dbpedia:Barack_Obama`) and a mention attribute, it returns all values for that attribute in mentions corresponding to that entity. By clicking on the number of mentions, all the mentions having that value for that attribute are returned, via an invocation of the first method.

- **Mention property occurrences**: given the URI of an entity (e.g., `dbpedia:Barack_Obama`), it returns all attributes used in mentions corresponding to that entity, with the number of mentions having that attribute. By clicking on the number of mentions, all the mentions having that attribute are returned, via an invocation of the first method.

Apart showing the feasibility of jointly querying data in different layers of the KnowledgeStore, these methods are also useful for assessing how selected entities were extracted from text, thus providing a concrete tool for spotting possible errors in the extraction process. Figure 12 shows the results of searching the `dbpedia:Fiat` entity.
Figure 12: Using reports to search the dbpedia:Fiat entity
4 The KnowledgeStore Architecture and Implementation

Changes wrt the KnowledgeStore Architecture described Deliverable D6.2.2

- updated Section 4.1 with the new ElasticSearch data store;
- updated software architecture (Section 4.2.1);
- added Section 4.3 on KnowledgeStore distribution;

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.

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

Client side The client side (upper part of Figure 13) 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 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;
- *linguistic processors* can also act as clients, by reading their input data from the KnowledgeStore and writing back the results of their computation;

35 http://newsreader.fbk.eu/knowledgestore
36 See http://www.w3.org/wiki/SparqlImplementations
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 13) 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 is used in combination with HBase to enforce the transactional guarantees of KnowledgeStore API operations (see Section 3.1);
- the ZooKeeper synchronization service is used to access and manage HBase nodes.

https://github.com/yahoo/omid
• 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;

• the ElasticSearch data store provides a database service for storing and querying semi-structured data; it can be used as an alternative of HBase and it does not need any additional software and/or server installed; in addition, it allows the user to install a stand-alone version of the KnowledgeStore, without the needing a multi-machine environment.

Not shown in Figure 13 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 4.1.4) and the KnowledgeStore Frontend Server (Section 4.1.5).

4.1.1 HBase & Hadoop

Hadoop[38] and HBase[39] 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 (HDFS) 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[40] data serializa-

[38]http://hadoop.apache.org
tion system. Benefits of this solution include space efficiency and transactional update of object values, as single-row operations are inherently transactional in HBase. Operations of the KnowledgeStore API may however affect multiple rows in different tables for each modified object, as 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\[41] which provides a full transaction manager over HBase. OMID exploits the versioning capabilities of HBase to realize a Multiversion Concurrency Control (MVCC) mechanism\[42] 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 14 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 14 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\[43]

4.1.2 ElasticSearch

Elasticsearch\[44] is a distributed, open source search and analytics engine, designed for horizontal scalability, reliability, and easy management. It builds distributed capabilities on top of Apache Lucene\[45], a Java-based indexing and search technology. Like HBase, Elasticsearch is distributed and scalable; it’s key features are search and analytics capabilities

\[41]\text{https://github.com/yahoo/omid/wiki}
\[42]\text{http://en.wikipedia.org/wiki/Multiversion_concurrency_control}
\[43]\text{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.}
\[44]\text{https://www.elastic.co/products/elasticsearch}
\[45]\text{https://lucene.apache.org/}
in real-time.

The use of ElasticSearch as an alternative to HBase has been investigated to allow the user to run an instance of the KnowledgeStore without the need of a multi-machine environment. As of October 2015, the ElasticSearch data store is available as a module in the KnowledgeStore, and tests have been performed on a single machine.

Table 1 shows some statistics on population of the WikiNews dataset (see Section 5.4.4) in text-only (compressed and uncompressed), HBase and ElasticSearch. The difference between HBase and ElasticSearch in population time may be caused by the different settings: while ElasticSearch is contained in a single machine, the HBase installation spreads over a cluster made by 4 machines.

Performances and tests of ElasticSearch on a cluster are under investigations.

4.1.3 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 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 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 physical file) can be set in the KnowledgeStore configuration and, depending on that, the performances of the HDFS may change (see Table 2).

---

Table 1: Disk usage of ElasticSearch on the WikiNews corpus

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Res. size (bytes)</th>
<th>Ment. size (bytes)</th>
<th>Pop. time (docs/h)</th>
<th>Count time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text-only</td>
<td>8134.34</td>
<td>352.23</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Text-only (gz)</td>
<td>671.97</td>
<td>45.35</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>HBase</td>
<td>9402.63</td>
<td>608.03</td>
<td>4,000</td>
<td>1.62</td>
</tr>
<tr>
<td>ElasticSearch</td>
<td>3645.79</td>
<td>319.06</td>
<td>60,000</td>
<td>0.02</td>
</tr>
</tbody>
</table>

---

46https://lucene.apache.org/
### 4.1.4 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 \( \langle \text{subject, predicate, object} \rangle \) 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 14, axiom metadata is not indexed and cannot thus be directly queried using SPARQL. The Open Source Edition of the Virtuoso triple store, version 7.0.0, has been chosen, motivated by its excellent performances in recent (April 2013) benchmarks 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

---


48 [http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V7/](http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V7/)
Table 2: 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).

<table>
<thead>
<tr>
<th>numSmallFile</th>
<th>totalAverageTime</th>
<th>individualAverageTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>13.3 s</td>
<td>40 ms/file</td>
</tr>
<tr>
<td>50</td>
<td>28.0 s</td>
<td>85 ms/file</td>
</tr>
<tr>
<td>100</td>
<td>45.7 s</td>
<td>138 ms/file</td>
</tr>
</tbody>
</table>

API using the Virtuoso Sesame driver. 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). 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.

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 \texttt{dbpedia:Volkswagen} as a \texttt{nwr:PublicCompany} and \texttt{nwr:PublicCompany} is a subclass of \texttt{nwr:Company} in the KnowledgeStore background knowledge, then a query for all companies (e.g., from the decision support suite) is expected to return \texttt{dbpedia:Volkswagen} as an answer. Although logical inference is a task for the second year of the project (T6.3, 49\footnote{http://www.openrdf.org/} \footnote{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.} \footnote{http://en.wikipedia.org/wiki/Readers-writer_lock} \footnote{The worst-case scenario repopulation is an expensive operation that may prevent SPARQL accesses for a long time (in the order of hours); therefore, this mechanism might be further refined in future releases of the KnowledgeStore, e.g., by repopulating from disk backups or using multiple instances of Virtuoso, one of which being always available for query answering while the others are synchronized.}
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.\footnote{Rule-based reasoning can be implemented through the fix-point evaluation of SPARQL queries.} 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 15 shows two examples of customized inference rules: rule in Figure 15a extends rule RDFS9 (the rule responsible for the \textit{dbpedia:Volkswagen} inference example above) and is applied on a per-context basis using TBox definitions (the \texttt{rdfs:subClassOf} triples) declared in a global context \texttt{ks:global}; rule in Figure 15b 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).

\subsection{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\footnote{http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/} standardized by W3C. The CRUD endpoint is instead implemented as an HTTP ReST service using JSON for Linked Data (JSON-LD)\footnote{http://json-ld.org/} as the data format. JSON-LD is a W3C proposed recommendation for encoding Linked Data in JSON, thus inheriting the tool support, readability characteristics and developer friendliness of the JSON format while being a concrete RDF syntax at the same time. The adoption of JSON-LD greatly improves the usability of the CRUD endpoint, allowing both RDF-aware as well as JSON-based applications (even dynamic web sites using Javascript / AJAX) to easily interact with the KnowledgeStore. HTTP authentication is used to implement the security requirements of the API, while HTTP compression supports the efficient transmission of JSON-LD data.

Internally, calls to the SPARQL endpoint are all forwarded to Virtuoso, while the majority of calls to the CRUD endpoint are forwarded to HBase & Hadoop, although \texttt{count()} and \texttt{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.

4.1.6 Suggested configurations and alternative backends

In the development of the KnowledgeStore, an aspect 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, we worked on different solutions, depending on the configuration of the user’s environment.

The following setting for a multi-machine setup is suggested:

• Hadoop as a file storage (see Section 4.1.1);
• HBase as the database for metadata and mentions (see Section 4.1.1); a possible cluster version of the data store using ElasticSearch is under investigation (see Section 4.1.2);
• Virtuoso (both community and commercial versions) as a triple store.

For a single-machine environment:

• Hadoop with a file:// local folder (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 4.1.3);
• ElasticSearch as the database for metadata and mentions (see Section 4.1.2);
• Virtuoso (especially the community version, free for a single machine setup) as a triple store.

To favour the test of the KnowledgeStore on a single machine by an end-user, during the development of the KnowledgeStore some additional modules have been investigated. They are still available, although the above setups are suggested.

• In place of HBase, a MySQL database or an Apache Lucene 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.

56 http://www.mysql.com/
57 https://lucene.apache.org/
4.2 Implementation

The implementation of the KnowledgeStore architecture described in Section 4.1 comprises two activities: (i) development of the software components specific to the KnowledgeStore, namely the KnowledgeStore Frontend Server and the Java Client library; and (ii) setup of the test and production deployment environments where all the server components of the system are integrated in a unifying framework. Software development is 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 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 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 according to Maven standards and easily importable in client applications 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 16. A Maven multi-module project organization has been adopted, with code organized in modules according to a functional criterion, as shown in Figure 17. 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:

http://maven.apache.org/
http://www.eclipse.org/
http://newsreader.fbk.eu/knowledgestore
**Code metrics reports**

![Code metrics report](image)

**Javadoc reference documentation**

![Javadoc documentation](image)

**Figure 16:** Examples of generated reports on the KnowledgeStore web site.

**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-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. Realizes also a file store sub-component that manages the files containing representations of resources (news, NLP annotations) and that implements the standard read, write and delete operations over files on top of Apache Hadoop HDFS version 1.0.4, exploiting the scalability and fault tolerance features that Hadoop provides.
ks-server-hbase  Realizes 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-server-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-server-http  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-elastic  Realizes a data store sub-component managing data about resources implemented using Elastic Search\footnote{http://www.elasticsearch.com/}.

ks-client  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.

ks-populator-naf  Implements the tool to populate the KnowledgeStore from documents in NAF\footnote{https://github.com/newsreader/NAF} format. See Section 5.1 for more information.

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

ks-tool  Contains some useful tools to dump the KnowledgeStore content in RDF format and to perform the performance tests described in \cite{Corcoglioniti et al., 2015b}.

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\footnote{https://www.virtualbox.org/} 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.

4.3 Distribution

The KnowledgeStore artifacts are published via the Maven central repository and on the download page\footnote{https://knowledgestore.fbk.eu/download.html} on the KnowledgeStore website. They include binaries, test binaries, sources and javadocs of each software module of the KnowledgeStore as well as the source and binary ‘tar.gz’ archives. The binary archives include all you need to run and populate a KnowledgeStore instance. They require a Java 7+ virtual machine, with startup scripts tested only on Linux and Mac OS X.
You can also watch the presentation video on YouTube\textsuperscript{65} or browse the sources and fork the project on our GitHub page\textsuperscript{66}

The \textit{KnowledgeStore} is distributed under the Apache License 2.0\textsuperscript{67}

\textsuperscript{65}http://youtu.be/YV0Qaljlta4
\textsuperscript{66}https://github.com/dkmfbk/knowledgestore
\textsuperscript{67}http://www.apache.org/licenses/LICENSE-2.0
5 The KnowledgeStore Population

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

Resource and mention data come from the NLP pipeline of WP4 and is expressed according to the 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.

As shown in figure 18, 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:

```xml
<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):

```plaintext
NewsReader: ICT-316404
February 8, 2016
```
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. 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 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, where its three main components (Buffer-Queue, Producers and Consumers) can be customized.

Figure 19 shows the internal architecture of the NAF-populator whose main components are:

- **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

---

Figure 19: NAF Multi-Threading populator.
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 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[^70]) 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.

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[^71]. 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:

[^70]: [http://newsreader.fbk.eu/knowledgestore](http://newsreader.fbk.eu/knowledgestore)

[^71]: 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.
• 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 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).

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. 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

---

72sort from GNU Core Utilities is available on different platforms, including Windows.
73Statistics 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.
Table 3: Data selection criteria.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linkability</td>
<td>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</td>
</tr>
<tr>
<td>focus on entities</td>
<td>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)</td>
</tr>
<tr>
<td>data relevance</td>
<td>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</td>
</tr>
<tr>
<td>focus on quality</td>
<td>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</td>
</tr>
<tr>
<td>common vocabularies</td>
<td>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</td>
</tr>
<tr>
<td>TBox inclusion</td>
<td>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</td>
</tr>
</tbody>
</table>

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 3. 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 4. They are all open-licensed and their entities are well interconnected by owl:sameAs links, although they use different, largely incompatible vocabularies.
Table 4: LOD datasets candidate for inclusion in the background knowledge.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Description</th>
<th>Availability</th>
<th>Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBpedia</td>
<td>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. <a href="http://dbpedia.org/">Lehmann et al., 2014</a></td>
<td>RDF dump, SPARQL, URI dereferencing</td>
<td>1.98B</td>
</tr>
<tr>
<td>Freebase</td>
<td>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. <a href="http://www.freebase.com/">Bollacker et al., 2008</a></td>
<td>RDF dump, URI dereferencing</td>
<td>2.4B</td>
</tr>
<tr>
<td>YAGO2</td>
<td>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. <a href="http://www.mpi-inf.mpg.de/yago-naga/yago/">Hoffart et al., 2013</a></td>
<td>RDF dump, URI dereferencing</td>
<td>120M</td>
</tr>
<tr>
<td>GeoNames</td>
<td>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. <a href="http://www.geonames.org/">http://www.geonames.org/</a></td>
<td>RDF dump, URI dereferencing</td>
<td>125M</td>
</tr>
<tr>
<td>LinkedGeoData</td>
<td>Geographic dataset automatically derived from OpenStreetMap with information about user-contributed points of interest (POIs) not covered by GeoNames. Linked to DBpedia and GeoNames. <a href="http://linkedgeodata.org">Stadler et al., 2012</a></td>
<td>RDF dump, REsT API</td>
<td>20B</td>
</tr>
<tr>
<td>Wikidata</td>
<td>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. <a href="http://www.wikidata.org/">Erxleben et al., 2014</a></td>
<td>unofficial, RDF dump (see citation)</td>
<td>475M</td>
</tr>
</tbody>
</table>
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.\(^{74}\) 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 terms (DCTERMS); Friend of a Friend (FOAF); Simple Knowledge Organization System (SKOS); WGS84 and GeoRSS\(^{75}\) 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 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 3 to narrow down the selection. As a result, the following DBpedia parts were selected for each language:\(^{76}\)

---

\(^{74}\)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.

\(^{75}\)http://www.w3.org/2005/Incubator/geo/XGR-geo/W3C_XGR_Geo_files/geo_2007.owl

\(^{76}\)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.

---

NewsReader: ICT-316404  February 8, 2016
• 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\(^{77}\) (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\(^{78}\) 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 described in Section 8. 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 20 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\(_{pro}\) tool described in Section 7. 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\(_{pro}\) 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

\(^{77}\)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.

\(^{78}\)http://patterns.dataincubator.org/book/smushing.html
Figure 20: Example of SPARQL query with (a) and without (b) smushing and inference.

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 PRO using a common format (Turtle Quads, i.e. N-Quads with Turtle encoding) 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 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

http://wiki.dbpedia.org/Internationalization/Guide
been measured on average, a number small enough to allow processing hundreds of millions of \texttt{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 \texttt{rdfpro} can produce the complete RDFS materialization of input data, we had to disable the inference rules (RDFS2 and RDFS3) on \texttt{rdfs:domain} and \texttt{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 \texttt{dbo:place} is also a \texttt{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 the form of VOID\footnote{http://www.w3.org/TR/void/} 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: \texttt{en}, \texttt{en\_ext}, \texttt{ml}, \texttt{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 2014 and DBpedia 2015 in September 2015, we reconfigured the pipeline and regenerated new versions of the datasets to be used in the next year. In the following, we provide some statistics about the processing done and the resulting datasets, covering the versions based on DBpedia 3.9 (tagged with \_39), the ones based on DBpedia 2014 (tagged with \_2014) and the last one (2015, tagged with \_2015) so that a comparison between the three can be made.

Table 5 reports the number of triples in each dataset, distinguishing between \texttt{rdf\_type} triples, \texttt{owl\_sameAs} triples, other ABox triples (essentially expressing entities properties) and TBox triples. Moving from DBpedia 2014 to DBpedia 2015 causes a small decrease of dataset size.

Table 6 reports the number of entities\footnote{Entities have been counted by selecting distinct URIs appearing as the subject of some \texttt{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.} 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.
Table 5: Number of triples in produced datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>rdf:type</th>
<th>owl:sameAs</th>
<th>ABox (other)</th>
<th>TBox</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>en_39</td>
<td>11 583 289</td>
<td>783 200</td>
<td>49 312 382</td>
<td>15 039</td>
<td>61 693 910</td>
</tr>
<tr>
<td>en_ext_39</td>
<td>13 239 452</td>
<td>783 200</td>
<td>61 153 352</td>
<td>15 041</td>
<td>75 191 045</td>
</tr>
<tr>
<td>ml_39</td>
<td>16 142 863</td>
<td>4 239 891</td>
<td>76 073 537</td>
<td>15 043</td>
<td>96 471 334</td>
</tr>
<tr>
<td>ml_ext_39</td>
<td>16 917 881</td>
<td>4 239 891</td>
<td>83 220 239</td>
<td>15 043</td>
<td>104 393 054</td>
</tr>
<tr>
<td>en_2014</td>
<td>15 184 293</td>
<td>859 158</td>
<td>59 774 033</td>
<td>19 798</td>
<td>75 837 282</td>
</tr>
<tr>
<td>en_ext_2014</td>
<td>17 279 671</td>
<td>859 158</td>
<td>77 751 702</td>
<td>19 816</td>
<td>95 910 347</td>
</tr>
<tr>
<td>ml_2014</td>
<td>23 881 931</td>
<td>4 691 477</td>
<td>94 079 675</td>
<td>19 810</td>
<td>122 672 893</td>
</tr>
<tr>
<td>ml_ext_2014</td>
<td>24 872 604</td>
<td>4 691 477</td>
<td>104 800 908</td>
<td>19 816</td>
<td>134 384 805</td>
</tr>
<tr>
<td>en_2015</td>
<td>13 138 091</td>
<td>910 725</td>
<td>59 268 670</td>
<td>19 718</td>
<td>73 337 204</td>
</tr>
<tr>
<td>en_ext_2015</td>
<td>15 539 931</td>
<td>1 097 251</td>
<td>78 436 030</td>
<td>19 735</td>
<td>95 092 947</td>
</tr>
<tr>
<td>ml_2015</td>
<td>22 378 634</td>
<td>4 981 271</td>
<td>88 542 689</td>
<td>19 746</td>
<td>115 922 340</td>
</tr>
<tr>
<td>ml_ext_2015</td>
<td>22 669 522</td>
<td>5 151 730</td>
<td>101 268 784</td>
<td>19 735</td>
<td>129 109 771</td>
</tr>
</tbody>
</table>

Table 6: Number of entities in produced datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>PER</th>
<th>ORG</th>
<th>GPELOC</th>
<th>FAC</th>
<th>PROD</th>
<th>WOA</th>
<th>EVENT</th>
<th>Misc</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>en_39</td>
<td>1 124 450</td>
<td>329 693</td>
<td>621 437</td>
<td>144 143</td>
<td>119 734</td>
<td>372 953</td>
<td>70 464</td>
<td>1 270 659</td>
<td>4 049 227</td>
</tr>
<tr>
<td>en_ext_39</td>
<td>1 246 076</td>
<td>344 802</td>
<td>666 932</td>
<td>165 266</td>
<td>123 814</td>
<td>399 611</td>
<td>94 049</td>
<td>1 292 297</td>
<td>4 299 465</td>
</tr>
<tr>
<td>ml_39</td>
<td>1 368 929</td>
<td>350 210</td>
<td>748 181</td>
<td>273 780</td>
<td>132 853</td>
<td>462 536</td>
<td>112 982</td>
<td>1 945 551</td>
<td>5 342 574</td>
</tr>
<tr>
<td>ml_ext_39</td>
<td>1 427 713</td>
<td>362 106</td>
<td>764 098</td>
<td>285 686</td>
<td>135 415</td>
<td>482 721</td>
<td>123 790</td>
<td>1 945 551</td>
<td>5 493 613</td>
</tr>
<tr>
<td>en_2014</td>
<td>1 649 672</td>
<td>302 727</td>
<td>849 711</td>
<td>164 446</td>
<td>128 196</td>
<td>389 118</td>
<td>89 216</td>
<td>1 226 617</td>
<td>4 634 402</td>
</tr>
<tr>
<td>en_ext_2014</td>
<td>1 673 873</td>
<td>324 190</td>
<td>915 939</td>
<td>187 701</td>
<td>133 324</td>
<td>424 830</td>
<td>115 921</td>
<td>1 306 946</td>
<td>4 863 236</td>
</tr>
<tr>
<td>ml_2014</td>
<td>2 069 158</td>
<td>352 411</td>
<td>1 275 738</td>
<td>423 843</td>
<td>144 297</td>
<td>499 760</td>
<td>220 940</td>
<td>2 056 611</td>
<td>6 006 109</td>
</tr>
<tr>
<td>ml_ext_2014</td>
<td>2 076 553</td>
<td>366 079</td>
<td>1 310 817</td>
<td>434 989</td>
<td>147 350</td>
<td>525 553</td>
<td>232 569</td>
<td>2 111 575</td>
<td>6 738 541</td>
</tr>
<tr>
<td>en_2015</td>
<td>2 135 045</td>
<td>220 422</td>
<td>596 509</td>
<td>143 052</td>
<td>119 393</td>
<td>347 496</td>
<td>80 663</td>
<td>584 115</td>
<td>4 225 234</td>
</tr>
<tr>
<td>en_ext_2015</td>
<td>2 164 052</td>
<td>252 859</td>
<td>668 125</td>
<td>178 130</td>
<td>135 244</td>
<td>426 225</td>
<td>116 485</td>
<td>670 048</td>
<td>4 564 208</td>
</tr>
<tr>
<td>ml_2015</td>
<td>2 669 196</td>
<td>275 377</td>
<td>787 692</td>
<td>413 737</td>
<td>142 034</td>
<td>471 046</td>
<td>310 106</td>
<td>1 393 960</td>
<td>6 446 328</td>
</tr>
<tr>
<td>ml_ext_2015</td>
<td>2 682 263</td>
<td>296 880</td>
<td>811 283</td>
<td>436 526</td>
<td>150 094</td>
<td>532 231</td>
<td>329 807</td>
<td>1 469 789</td>
<td>6 661 598</td>
</tr>
</tbody>
</table>

NewsReader: ICT-316404 February 8, 2016
Table 7: Processing statistics.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Files</th>
<th>Output triples</th>
<th>Execution time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Download</td>
<td>Filtering</td>
</tr>
<tr>
<td>en,39</td>
<td>17</td>
<td>77770.116</td>
<td>70.268.829</td>
</tr>
<tr>
<td>en_ext,39</td>
<td>85</td>
<td>139812.619</td>
<td>110394.636</td>
</tr>
<tr>
<td>ml,39</td>
<td>43</td>
<td>178654.555</td>
<td>120011.745</td>
</tr>
<tr>
<td>ml_ext,39</td>
<td>98</td>
<td>219324.567</td>
<td>145968.914</td>
</tr>
<tr>
<td>en,2014</td>
<td>17</td>
<td>98897.879</td>
<td>82371.053</td>
</tr>
<tr>
<td>en_ext,2014</td>
<td>85</td>
<td>176899.005</td>
<td>133811.216</td>
</tr>
<tr>
<td>ml,2014</td>
<td>43</td>
<td>236496.207</td>
<td>144612.632</td>
</tr>
<tr>
<td>ml_ext,2014</td>
<td>98</td>
<td>287518.712</td>
<td>177634.142</td>
</tr>
<tr>
<td>en,2015</td>
<td>17</td>
<td>75875.870</td>
<td>65707.375</td>
</tr>
<tr>
<td>en_ext,2015</td>
<td>85</td>
<td>142990.109</td>
<td>102832.650</td>
</tr>
<tr>
<td>ml,2015</td>
<td>43</td>
<td>202306.201</td>
<td>115089.945</td>
</tr>
<tr>
<td>ml_ext,2015</td>
<td>98</td>
<td>242809.561</td>
<td>135211.128</td>
</tr>
</tbody>
</table>

that could not be classified under previous types. As the classification of entities is not exclusive in DBpedia, the total value does not represent the sum over the individual entity types.

Table 7 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 21, and can help in understanding and using the dataset, e.g., by supporting the construction of SPARQL queries.

---

82 Classification according to the types of the annotation guidelines has been performed on the basis of 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 21: Examples of browsing the statistics ontology in Protégé.
5.4 The KnowledgeStore in action: use cases

The KnowledgeStore was successfully deployed, populated, and exploited to build enhanced applications in several concrete NewsReader scenarios, described next. In particular, we will 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 and a cross-resource processor, the VUA Event Coreference Module.

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, 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 [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

http://ixa2.si.ehu.es/nrdemo/demo.php
http://ic.vupr.nl/~ruben/vua-eventcoreference.ttl/
http://fracor.bitbucket.org/rdfpro/
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 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, a visual analytics application that delivers real time interaction with dynamic network-centric data. Synerscope interacts with the KnowledgeStore through the KnowledgeStore exporter, 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) 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. The entity layer of the KnowledgeStore was also populated with 270 millions of statements coming from a selected multi-lingual subset of DBpedia. 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

86 http://www.synerscope.com/
87 Made available for project purposes by LexisNexis - www.lexisnexis.nl/
88 RDF data produced by the VUA Event Coreference Module is publicly available at http://datahub.io/dataset/global-automotive-industry-news
89 The background knowledge dataset used is available for download at http://knowledgestore.fbk.eu.
required significant time and effort, the development of applications on top of stored contents was realized as part of a Hack Day event,\footnote{http://www.newsreader-project.eu/newsreader-world-cup-hack-day/} 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**\footnote{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.} 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\footnote{Accessible at: https://newsreader.scraperwiki.com, code available at https://bitbucket.org/scraperwikids/newsreader_api_flask_app} [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.\footnote{The background knowledge dataset used is available for download at http://knowledgestore.fbk.eu.} 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 through 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.

\textsuperscript{90}http://www.newsreader-project.eu/newsreader-world-cup-hack-day/\footnote{held in London, 10th of June 2014.} \textsuperscript{91}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. \textsuperscript{92}Accessible at: https://newsreader.scraperwiki.com, code available at https://bitbucket.org/scraperwikids/newsreader_api_flask_app. \textsuperscript{93}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 crisis, focused on the automotive industry sector. 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 been applied (Section 8) to get new information inferred from the events extracted by the pipeline.

Clients: Applications The main applications of this dataset are the two 2015 hackathons (January 21st in Amsterdam and January 30th in London). The events were of interest 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 million mentions, resulting in around 100 million events extracted by the VUA Event Coreference Module, linking to DBpedia ~652K persons, ~873K organisations and ~274K locations. Using the ESO reasoner 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).

At the time of writing this deliverable (October 2015), we are populating an additional version of the Global Automotive Industry dataset (version 3), covering the whole period 01.2003-10.2015 and consisting of more than 2M news articles.

5.4.4 Scenario 4: Wikinews

This KnowledgeStore instance was populated with 18,510 general domain news from Wikinews (news time period: 2004-2014).

Given its controlled size, substantially smaller than the other KnowledgeStore populations here reported, this instance is used project-wise to benchmark and improve the performances of the NewsReader information extraction processors, by comparing their

95 http://www.newsreader-project.eu/hackathon-newsreader-london-jan-30-2015/
96 http://en.wikinews.org
outputs with gold standard annotations produced by a team of linguists as part of the project.

Furthermore, as the source news are publicly available[97] this allows us to make available through a publicly accessible KnowledgeStore instance[98] a complete dataset consisting of structured content (mentions, entities, axioms) linked to the source news from which it was extracted, thus favoring the dissemination of the project results and enabling other researchers and developers to exploit this content for various purposes (e.g., benchmarking their information extraction pipelines, building and testing new LOD applications).

5.4.5 Scenario 5: The Dutch House of Representatives

In this scenario, a KnowledgeStore instance was populated with content extracted from parliamentary inquiry material provided by the Dutch House of Representatives. The aim of the scenario was to make the information around the parliamentary inquiry on the financial system that was instigated by the Dutch House of Representatives in November 2010 more insightful.

The Information Provision Department of the Dutch House of Representatives provided us with 826,964 documents they used in this parliamentary enquiry from their archive. This set consisted of news articles, magazine articles, debate transcripts and parliamentary papers. These documents were cleaned and documents that were too short were discarded. The remaining 627,341 documents were processed with the NewsReader Dutch pipeline v1. From the LexisNexis archives, we added 57,382 specifically about ABN-AMRO, one of the main players in this use case. These were processed using the same pipeline.

17,583,997 mentions were annotated in the text, from which 5,383,498 event instances and 188,296,316 triples were extracted. In addition, 122,672,893 triples from a multilingual version of the DBpedia background Knowledge (ml \textsuperscript{2014} in Table 5) were uploaded into the KnowledgeStore.

In June 2015, the navigation of the KnowledgeStore via the Synerscope tool was presented to about 10 members of the Information Provision Department of the Dutch House of Representatives (amongst which the head of the department), as well as three members of De Nederlandsche Bank (The Dutch Bank) who had expressed their interest in this use case.

5.5 Discussion

Two years 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.

[97] Under the terms of the Creative Commons Attribution 2.5 License.
[98] https://knowledgestore2.fbk.eu/nwr/wikinews/
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 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) paradigm, possibly together with specification languages that users can exploit to formulate their analysis, such as the procedural data-flow language Pig Latin or the declarative, SQL-based language HiveQL. 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. 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

---

99 en.wikipedia.org/wiki/Bulk_synchronous_parallel
100 http://pig.apache.org/
101 https://hive.apache.org/
102 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.

NewsReader: ICT-316404 February 8, 2016
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 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, 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.

5.6 Statistics extraction on a populated KnowledgeStore instance

We developed a bunch of tools to compute significant statistics on a running KnowledgeStore installation. In particular, Linking Analyzer produces an indented HTML file where, for each DBpedia Ontology class, the number (and percentage over the total) of instances of that class having some mention in a document is reported, as well as the number (and percentage over the total) of documents containing mentions of instances of that class. An example (excerpt) of output produced by the tool is reported in Figure 22.

KS Content Report runs arbitrary analytic queries against a running KnowledgeStore instance, to obtain information such as the most frequent FrameNet frame extracted in the news, the most cited DBpedia person, organization, and location in the news, etc. Currently, the tool performs 87 different queries, available under the configuration folder of the source code. An example (excerpt) of the output produced by running the mostFrequentFrameElements query of KS Content Report is shown in Table 8. This set of queries can be easily extended to perform additional analysis.
Table 8: The `mostFrequentFrameElements` query on the `KnowledgeStore`.

<table>
<thead>
<tr>
<th>FrameElement</th>
<th>Occurences</th>
</tr>
</thead>
<tbody>
<tr>
<td>framenet:Statement@Speaker</td>
<td>22553</td>
</tr>
<tr>
<td>framenet:Choosing@Cognizer</td>
<td>12802</td>
</tr>
<tr>
<td>framenet:Text_creation@Author</td>
<td>12446</td>
</tr>
<tr>
<td>framenet:Request@Speaker</td>
<td>3374</td>
</tr>
<tr>
<td>framenet:Possession@Owner</td>
<td>2948</td>
</tr>
<tr>
<td>framenet:Request@Addressee</td>
<td>2566</td>
</tr>
<tr>
<td>framenet:Arriving@Theme</td>
<td>2299</td>
</tr>
<tr>
<td>framenet:Bringing@Theme</td>
<td>2265</td>
</tr>
<tr>
<td>framenet:Causation@Cause</td>
<td>2235</td>
</tr>
<tr>
<td>framenet:Bringing@Agent</td>
<td>1975</td>
</tr>
<tr>
<td>framenet:Attempt@Agent</td>
<td>1900</td>
</tr>
<tr>
<td>framenet:Statement@Topic</td>
<td>1811</td>
</tr>
<tr>
<td>framenet:Telling@Speaker</td>
<td>1781</td>
</tr>
<tr>
<td>framenet:Telling@Addressee</td>
<td>1772</td>
</tr>
<tr>
<td>framenet:Removing@Agent</td>
<td>1709</td>
</tr>
<tr>
<td>framenet:Removing@Cause</td>
<td>1708</td>
</tr>
<tr>
<td>framenet:Statement@Message</td>
<td>1705</td>
</tr>
<tr>
<td>framenet:Using@Agent</td>
<td>1652</td>
</tr>
<tr>
<td>framenet:Taking@Theme</td>
<td>1637</td>
</tr>
<tr>
<td>framenet:Getting@Recipient</td>
<td>1636</td>
</tr>
<tr>
<td>type</td>
<td># linked instances</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Agent</td>
<td>20547</td>
</tr>
<tr>
<td>Organisation</td>
<td>6763</td>
</tr>
<tr>
<td>Company</td>
<td>3281</td>
</tr>
<tr>
<td>RecordLabel</td>
<td>117</td>
</tr>
<tr>
<td>PoliticalParty</td>
<td>126</td>
</tr>
<tr>
<td>Band</td>
<td>447</td>
</tr>
<tr>
<td>EducationalInstitution</td>
<td>1051</td>
</tr>
<tr>
<td>University</td>
<td>837</td>
</tr>
<tr>
<td>GovernmentAgency</td>
<td>218</td>
</tr>
<tr>
<td>SportsTeam</td>
<td>483</td>
</tr>
<tr>
<td>SoccerClub</td>
<td>405</td>
</tr>
<tr>
<td>Broadcaster</td>
<td>272</td>
</tr>
<tr>
<td>TelevisionStation</td>
<td>135</td>
</tr>
<tr>
<td>TradeUnion</td>
<td>36</td>
</tr>
<tr>
<td>Non-ProfitOrganisation</td>
<td>96</td>
</tr>
<tr>
<td>Legislature</td>
<td>47</td>
</tr>
<tr>
<td>SportsLeague</td>
<td>118</td>
</tr>
<tr>
<td>MilitaryUnit</td>
<td>119</td>
</tr>
<tr>
<td>Person</td>
<td>13645</td>
</tr>
<tr>
<td>OfficeHolder</td>
<td>791</td>
</tr>
<tr>
<td>Athlete</td>
<td>5861</td>
</tr>
<tr>
<td>RacingDriver</td>
<td>790</td>
</tr>
<tr>
<td>FormulaOneRacer</td>
<td>214</td>
</tr>
<tr>
<td>NASCARDriver</td>
<td>354</td>
</tr>
<tr>
<td>SoccerPlayer</td>
<td>378</td>
</tr>
<tr>
<td>GridironFootballPlayer</td>
<td>637</td>
</tr>
<tr>
<td>AmericanFootballPlayer</td>
<td>555</td>
</tr>
<tr>
<td>Wrestler</td>
<td>116</td>
</tr>
<tr>
<td>Artist</td>
<td>1643</td>
</tr>
<tr>
<td>MusicalArtist</td>
<td>1055</td>
</tr>
<tr>
<td>Writer</td>
<td>230</td>
</tr>
<tr>
<td>FictionalCharacter</td>
<td>536</td>
</tr>
<tr>
<td>SoapCharacter</td>
<td>158</td>
</tr>
<tr>
<td>ComicCharacter</td>
<td>109</td>
</tr>
<tr>
<td>Politician</td>
<td>460</td>
</tr>
<tr>
<td>Senator</td>
<td>58</td>
</tr>
<tr>
<td>Royalty</td>
<td>148</td>
</tr>
<tr>
<td>BritishRoyalty</td>
<td>148</td>
</tr>
<tr>
<td>Scientist</td>
<td>192</td>
</tr>
<tr>
<td>Cleric</td>
<td>126</td>
</tr>
<tr>
<td>ChristianBishop</td>
<td>48</td>
</tr>
</tbody>
</table>

Figure 22: Example of output obtained running Linking Analyzer against a populated KnowledgeStore instance.
6 Performance assessment

In this section we report the results of a number of experiments aimed at assessing the performances of the KnowledgeStore. We focus on two core operations that are relevant for the practical adoption of the system: data population, analyzed in Section 6.1, and data retrieval, analyzed in Section 6.2. All the experiments use real world data from the Cars (Ver. 2) dataset (Section 5.4.3) and were conducted on a cluster of five servers connected by a Gigabit LAN: one running the KnowledgeStore Server and the Virtuoso triplestore and having 1 TB disk, 32 GB RAM and two Intel Xeon E5-2440 CPUs; the others running Hadoop and HBase and each having 1 TB disk, 32 GB RAM (except a server with 8 GB only) and two Intel Xeon E6545 CPUs.

6.1 Data population performances

As previously discussed, the KnowledgeStore offers a number of data upload (and manipulation) operations that support different population workflows. Here we consider the workflow adopted in NewsReader and described in Section 5, in which both the resource and mention layers are populated first (e.g., using the NAF populator) with the results of the single-document information extraction processors; then, the entity layer is populated (e.g., using rdf_pro, see Section 7) with the entities and axioms extracted by the cross-document information extraction processors, as well as with background knowledge. These two steps have very different performances: while the entity layer population is fast—400K axioms/s, corresponding to 4M news articles per hour given an average of 350 axioms per news article (Cars (Ver. 2) dataset)—the population of the resource and mention layers is around three order of magnitude slower—∼8K news articles per hour—and thus dominates and determines the overall population performances.

We now concentrate on the performances of the population of the resource and mention layers alone, and investigate through a couple of experiments whether and how they are impacted by the average resource size (in particular, number of mentions per news article) and overall dataset size (number of resources).

Impact of resource size. The size of a news article can be expressed in many ways: e.g., file size, number of metadata attributes, number of words, number of mentions it contains. Among them, the number of mentions is the property most impacting on the performances of the population process, as each mention must be inserted in HBase and linked to the

---

Footnote: We ignored the time needed for populating the background knowledge, whose size does not depend on the number of news articles populated. In fact, with a rate of 400K axioms/s, storing a representative background knowledge dataset of 100M DBpedia axioms takes only 4 minutes, which is negligible if compared to the total population time measured in the order of hours.
Table 9: Population time and rate depending on number of mentions per news article.

<table>
<thead>
<tr>
<th>Batch</th>
<th>Avg. mentions per news article</th>
<th>Population time [h]</th>
<th>Population rate [news articles/h]</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>44</td>
<td>0.51</td>
<td>11,651</td>
</tr>
<tr>
<td>large</td>
<td>300</td>
<td>1.32</td>
<td>4,537</td>
</tr>
</tbody>
</table>

Table 10: Population rate depending on amount of data already stored. We recall that Resources include both the original news articles as well as its NAF annotated version (i.e., the number of resources is twice the number of news articles).

<table>
<thead>
<tr>
<th>Resources Mentions Disk space [GB]</th>
<th>Population rate [news articles/h]</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>7,537</td>
</tr>
<tr>
<td>1,239,270 105,674,491 125</td>
<td>7,872</td>
</tr>
<tr>
<td>2,296,874 189,360,192 224</td>
<td>7,830</td>
</tr>
</tbody>
</table>

corresponding news article and entity, and in the KnowledgeStore implementation these operations are more expensive than storing the news article file.

**Impact of dataset size.** A degradation of the population rate can be reasonably expected as the amount of data stored in the KnowledgeStore increases and storing and indexing new data become more expensive. We thus conducted another experiment to assess the extent of this phenomenon and, consequently, the scalability of the population process with large dataset sizes. We populated the resource and mention layers of the KnowledgeStore under test with the whole Cars (Ver. 2) dataset, which consists of ~1.3M news articles (for a total of 2.6M resources, including the corresponding NAF annotated versions) each one having on average 163 mentions (standard deviation = 58). We measured the (instantaneous) population rate at the beginning, in the middle and around the end of the population process, and took note of the state of the system when the three measurements were made, which consists of the number of resources and mentions already stored and the total disk space used in the cluster. The results are reported in Table 10.

The three rates measured are similar and do not exhibit any clear trend, thus suggesting that the population rate can be considered roughly constant during the whole population process. This finding is consistent with the performance characteristics of the technologies used—especially HBase and Hadoop HDFS—with small differences in rate imputable to minor differences in the populated news articles or in the occasional triggering of background maintenance processes in HDFS and HBase (e.g., HBase table compaction). Although the results of the experiment cannot be generalized to datasets bigger than the one considered (we expect major degradation and eventual failure when approaching the storage capacity of the Hadoop cluster), they show nevertheless that consistent population
performances can be achieved given the software infrastructure the KnowledgeStore builds on.

### 6.2 Data retrieval performances

In this section we assess the performances of the data retrieval operations offered by the KnowledgeStore (SPARQL queries and resource, mention, and file retrieval) with different dataset sizes and numbers of concurrent clients.

To this purpose, we populated the KnowledgeStore under test with different test datasets of increasing size but similar schema and characteristics, all obtained from the Cars (Ver. 2). We then selected a set of parametric retrieval requests that are representative of possible interactions of the user with the system for these datasets, starting from the queries of the NewsReader Simple API. We call request mix the instantiation of these requests for a specific set of parameter values and define the evaluation of a request mix as the sequential evaluation of the requests of the mix. For each test dataset and different numbers of clients (independent variables) we simulated the concurrent evaluation by these clients of a large number of request mixes, and we measured the overall request throughput and the average request evaluation time (dependent variables).

In the following, we describe in details the test datasets, the parametric requests and the test procedure we adopted, and report and discuss the obtained results. While the test datasets are copyrighted and cannot be made publicly available, the test tools used and their configuration are both available on the KnowledgeStore website and can be used to perform similar evaluations with different datasets or parametric requests (in particular, they can be used on the publicly available Wikinews KnowledgeStore instance).

**Test datasets.** We started from the Cars (Ver. 2) dataset due to its real-world contents and large size (∼1.3M news articles / ∼2.6M resources), and we built five test datasets of increasing size (D1 to D5) by selecting only specific subsets of the source dataset. Table 11 describes the obtained datasets. The scale factor, computed as the ratio of the numbers of resources in different datasets, provides an indication of the relative dataset sizes. They have been chosen to roughly follow a logarithmic scale, with deviations caused by the practical need to base the selection on the available news article batches forming the source dataset.

**Parametric requests.** Table 12 reports the name and informal description of the 14 parametric requests selected for the test (parameters are emphasized in the description). These parametric requests derive from the ones of the NewsReader Simple API [Hopkinson et al., 2014], an API built on top of the KnowledgeStore to serve practical data analysis needs and used by data analysts and journalists during the Hack Day events (Sections 5.4.2, 5.4.3).

Overall, the parametric requests and their sequential evaluation within a request mix simulate the typical activities of a user exploring the dataset:

107 https://knowledgestore.fbk.eu/test-tools.html
Table 11: The test datasets (from smaller to larger, in terms of number of resources).

<table>
<thead>
<tr>
<th>Scale factor</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resources</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>News Articles</td>
<td>163,752</td>
<td>341,464</td>
<td>572,690</td>
<td>1,239,270</td>
<td>2,519,496</td>
</tr>
<tr>
<td>NAF Documents</td>
<td>81,876</td>
<td>170,732</td>
<td>286,345</td>
<td>619,635</td>
<td>1,259,748</td>
</tr>
<tr>
<td>Mentions</td>
<td>14,031,629</td>
<td>28,836,259</td>
<td>48,329,826</td>
<td>105,674,491</td>
<td>205,114,711</td>
</tr>
<tr>
<td>Entities</td>
<td>2,052,664</td>
<td>4,159,978</td>
<td>6,285,449</td>
<td>13,998,806</td>
<td>25,156,574</td>
</tr>
<tr>
<td>Events</td>
<td>1,829,866</td>
<td>3,752,010</td>
<td>6,285,449</td>
<td>13,998,806</td>
<td>25,156,574</td>
</tr>
<tr>
<td>Persons</td>
<td>81,265</td>
<td>152,951</td>
<td>241,774</td>
<td>446,356</td>
<td>729,797</td>
</tr>
<tr>
<td>Organizations</td>
<td>94,693</td>
<td>177,171</td>
<td>281,526</td>
<td>540,077</td>
<td>947,262</td>
</tr>
<tr>
<td>Locations</td>
<td>46,840</td>
<td>77,846</td>
<td>111,837</td>
<td>187,631</td>
<td>290,091</td>
</tr>
<tr>
<td>Axioms (Triples)</td>
<td>128,035,674</td>
<td>159,180,410</td>
<td>201,090,830</td>
<td>322,981,854</td>
<td>535,035,576</td>
</tr>
<tr>
<td>from Mentions</td>
<td>32,060,740</td>
<td>63,205,476</td>
<td>105,115,896</td>
<td>227,006,920</td>
<td>439,060,642</td>
</tr>
<tr>
<td>from Background Knowledge</td>
<td>95,974,934</td>
<td>95,974,934</td>
<td>95,974,934</td>
<td>95,974,934</td>
<td>95,974,934</td>
</tr>
<tr>
<td>Total Disk Space (GB)</td>
<td>30</td>
<td>50</td>
<td>76</td>
<td>152</td>
<td>280</td>
</tr>
</tbody>
</table>

Table 12: Parametric requests used in the test.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sparql 1</td>
<td>Get first 20 DBpedia actors with given type and name, sorted by number of participated events.</td>
</tr>
<tr>
<td>sparql 2</td>
<td>Get value of selected property for first 20 actors with given type and name, sorted by value.</td>
</tr>
<tr>
<td>sparql 3</td>
<td>Get event labels matching a term with number of associated events, in decreasing order.</td>
</tr>
<tr>
<td>sparql 4</td>
<td>Get summaries of first 20 events having specific year and actor, sorted by event date.</td>
</tr>
<tr>
<td>sparql 5</td>
<td>Get summaries of first 20 events having specific year and pair of actors, sorted by event date.</td>
</tr>
<tr>
<td>sparql 6</td>
<td>Get summaries of first 20 events having specific year and actor type, sorted by event date.</td>
</tr>
<tr>
<td>sparql 7</td>
<td>Get summaries of first 20 events having specific year and label term, sorted by event date.</td>
</tr>
<tr>
<td>sparql 8</td>
<td>Get summaries of first 20 events having specific year, type and label term, sorted by event date.</td>
</tr>
<tr>
<td>sparql 9</td>
<td>Get all the information available for a given event (incl. mention URIs).</td>
</tr>
<tr>
<td>sparql 10</td>
<td>Get all the information available for a given actor (incl. mention URIs).</td>
</tr>
<tr>
<td>sparql 11</td>
<td>Get first 20 persons related to a given actor, sorted by number of events in common (at least one).</td>
</tr>
<tr>
<td>sparql 12</td>
<td>Get details of first 20 events participated by a given actor, sorted by event URI.</td>
</tr>
<tr>
<td>crud 1</td>
<td>Get the metadata of a given resource and the attributes of all the mentions it contains.</td>
</tr>
<tr>
<td>crud 2</td>
<td>Download the textual content of a given resource.</td>
</tr>
</tbody>
</table>
• the user searches for events based on certain properties, such as event year, type, term, involved actor(s) URIs, and actor types (requests `sparql 4` to `sparql 8`); in order to constrain these properties, the user may have first to search for a specific actor (`sparql 1`, `sparql 2`) or get an idea of what event terms are in the dataset (`sparql 3`);
• the user then selects an event and retrieves all the information about it (`sparql 9`);
• the user selects an event actor and gets the corresponding description (`sparql 10`), including other persons related to the actor (`sparql 11`) and all the events the actor participates in (`sparql 12`);
• the user chooses a resource mentioning the selected event and retrieves its text (`crud 2`), metadata and mentions (`crud 1`), i.e., the information needed to build a visualization such as the one of Figure [11a

Test procedure. A single test consists in the evaluation of randomly selected request mixes by one or more clients for a fixed period of time, after which performance metrics are produced; clients operate concurrently but each client submits its requests sequentially (as a user would do).

For each dataset, multiple tests were performed according to the following test procedure:

1. the dataset is loaded into the KnowledgeStore;
2. 1M request mixes is generated with a random choice of parameters, for later use in the tests;
3. the KnowledgeStore is restarted, so to begin from a clean state;
4. a warmup test with 24 clients is run for 45 minutes, discarding its results; the warmup allows for the initialization of the system and its caches, leading to a steady, optimal performance state;
5. the test with one client is run for 90 minutes, a time large enough to perform a number of query mixes comparable to the one of the other tests;
6. the tests with 2, 3, 4, 6, 8, 12, 16, 24, 32, 48 and 64 clients are run sequentially, 30 minutes each.

To support this procedure two specific tools have been developed and made available on the KnowledgeStore web site: (i) the query test generator tool produces an arbitrary number of request mixes by sampling and joining the results of auxiliary queries that extract the admissible parameter values; and, (ii) the query test driver takes the produced request mixes and performs a single test as described above, recording several performance figures for later analysis.

Test results. The two line charts of Figure [23] show respectively the throughput measured in request mixes per hour (a), and the average request evaluation time (b) as functions of the number of concurrent clients, with a line for each test dataset.

As one could expect, adding new clients determines an increase of throughput with minor changes of the evaluation time up to a certain threshold, after which all the physical resources of the system (mainly CPU cores) are saturated, the throughput remains
Figure 23: Request throughput (a) and average evaluation time (b) with different clients and dataset sizes.

(almost) constant, and the evaluation time increases linearly as requests are queued for later evaluation. In the system under test the threshold is located around 12 clients (vertical lines in the charts), a quantity that matches the number of CPU cores available to the Virtuoso triplestore. This correspondence is explained by the fact that the majority of parametric requests are SPARQL queries that end up hitting Virtuoso. Nevertheless, the request mixes also include CRUD requests that ultimately hit the HBase and Hadoop HDFS clusters and may scale well beyond 12 clients; this fact likely explains the slight increase in throughput after the 12 clients’ threshold for the smallest datasets where the performances of Virtuoso impact less.

While quantifying precisely the effect of the dataset size on retrieval performances is difficult, as there are many notions of ‘size’ to account for (number of news articles, resources, axioms, entities), it is interesting to note that a ~15 times increase in the number of news articles, from $D1$ (81K news articles) to $D5$ (1.3M news articles), caused ‘only’ a ~2 times decrease in the throughput, from 21,126 to 10,212 requests/h for 64 clients. As the evaluation is made on real-world data, this finding is particularly significant for the practical adoption of the system. The line charts also feature local throughput maxima as well as global evaluation time minima (around 4/6 clients): these features cannot be ascribed to noisy test conditions or measurements and their explanation requires further investigation.

The bar chart of Figure 24 shows, for each parametric request, the average evaluation times for the different test datasets. From the graph it is clear that some parametric

\[\text{We suspect that possible causes include (but may not be limited to) the better/worse use of caches at various levels of the system and increased synchronization overhead beyond a certain level of concurrency.}\]
requests are much more expensive than others and the performances of some requests (especially the slowest SPARQL queries) degrade more markedly with an increase in dataset size. An analysis of the four most expensive parametric requests (sparql 6, sparql 3, sparql 8, sparql 7) shows that even if they return few results (due to the use of the SPARQL LIMIT clause) they all present characteristics typical of analytical queries: they are not selective and thus match, join, sort and aggregate large amounts of entity data. For instance, sparql 6 has to consider all the events of a certain year (in the order of millions) and all the instances of a given actor type (hundreds of thousands), join them based on the participation relation and sort the results (possibly millions of tuples) by date to return only the first 20. On the other end, the least expensive queries sparql 4, sparql 12, sparql 9, sparql 11 and sparql 5 are much more selective: sparql 9 is the lookup of a single event, while the other queries consist essentially in the lookup of the events related to a specific actor, whose number is limited and largely independent of the dataset size. In terms of evaluation time, CRUD retrieval operations are situated halfway; while crud 1 (resource and mention retrieval hitting HBase) is slower with larger datasets, the performances of crud 2 (file download hitting HDFS) are largely independent of the dataset size.
7 The RDFPRO Tool

Changes wrt the KnowledgeStore Deliverable D6.2.2

- added RDFPRO web interface (Section 7.5);
- added new processors, rules and mapreduce (Section 7.2);

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 RDFPRO (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 RDFPRO 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 RDFPRO model in Section 7.1, its processors in Section 7.2 and its implementation in Section 7.3. Finally, Section 7.4 describes the evaluation and Section 7.5 contains an essential documentation on how to use the tool.

7.1 Processing model

The processing model of RDFPRO is centered around the concept of RDF processor. A processor @ (Figure 25a) is a software component that consumes an input stream of RDF quads—i.e., RDF triples with an optional fourth named graph component—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

---

109 Processors are denoted by “@” in RDFPRO syntax.
110 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 RDFPRO on plain triple data.
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 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 RDFpro (Section 7.2) or implemented by users, new pipeline processors can be derived by (recursively) applying sequential and parallel compositions. In a sequential composition (Figure 25b), 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 25c), 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 25d: 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 \( @\text{read} \) and \( @\text{write} \) 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 RDFpro tool relies on these processors for all I/O, ignoring global input and output streams that are instead accessible when using RDFpro as a library).

![Figure 25: Processor (a); sequential (b) & parallel (c) composition; example (d) – full syntax on web site.](image-url)
7.2 Processors

In order to address the processing tasks considered at the beginning of this section, we implemented the following processors in RDFPro:

@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 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. 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.

@rules  Emits the closure of input quads using a customizable set of rules. Rules heads and bodies are SPARQL graph patterns, with FILTER, BIND, and UNION constructs allowed in the body. The current implementation is based on Drools.

---

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

112 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}

113 Drools is a rule engine implementing the RETE algorithm. See http://www.drools.org/
@mapreduce Applies a custom map script (JavaScript or Groovy) to label and group input quads into partitions, each one reduced with a reduce script. A multi-threaded, non-distributed MapReduce implementation based on the sort primitive is used.

@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”.

7.3 Implementation

RDFPro is implemented in Java on top of the open source Sesame RDF library[114] 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[115] 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

[115] 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.
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 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.

7.4 Empirical Evaluation

To answer our research question, we perform an empirical evaluation of RDFpro in four broad, relevant usage scenarios that exemplify the considered RDF processing tasks. In the first three scenarios—dataset analysis (Section 7.4.1), filtering (Section 7.4.2) and merging (Section 7.4.3)—we conduct practical experiments using a commodity workstation[116] 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 7.4.4)—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 RDFpro website[117].

7.4.1 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

---

[116] Intel Core i7 860 CPU (4 cores, hyper-threading), 16 GB RAM, 500 GB 7200 RPM hard disk, Linux 2.6.32.

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

We use the \texttt{@tbox} and \texttt{@stats} processors to extract TBox and VOID statistics, invoked both separately and aggregated in a pipeline processor as shown in Figure 26a. To extract new triples, we read both dataset releases and use parallel composition with the \texttt{difference} set operator (Section 7.1) to combine quads, as shown in Figure 26b.

Table 26c 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 \texttt{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 \texttt{RDFpro} streaming model and composition facilities, can generally lead to better performance due to a reduction of I/O overheads.

7.4.2 Dataset Filtering

When dealing with large RDF datasets, dataset filtering (or slicing \cite{Marx2013}) is often required to extract a small subset of interesting data, identified, e.g., based on a previous dataset analysis (Section 7.4.1). 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 \texttt{fb:music.musical_group}) that are still active (i.e., there is no associated property \texttt{fb:music.artist.active.end}), and the properties to extract are the
We implement the task with two invocations of RDF\textsubscript{PRO} as in Figure 27a. 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 \texttt{@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 \texttt{@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 \texttt{@transform} expression).

Table 27b reports the execution times, throughputs, input and output sizes of the two 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\textsubscript{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 7.4.5).

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\textsubscript{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).
Figure 27: Dataset filtering flow (a) and results (b).

In cases where RDF<sub>pro</sub> 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.

### 7.4.3 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 28a 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 \rangle 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 28b 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 RDFpro (upper part of the table), or aggregated per processing phase using composition capabilities (lower part). Additionally, RDFpro reported the use of \sim 2 GB of memory for smushing an owl:sameAs graph of \sim 38M URIs and \sim 8M equivalence classes (\sim 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 RDFpro website\[119\].

\[119\] http://fracor.bitbucket.org/rdfpro/
7.4.4 Dataset Massaging

We categorize three relevant, broad classes of dataset massaging tasks that are supported by RDF 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 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. 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. 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).

7.4.5 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

\[120\] 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.

\[121\] E.g., the OWLIM Lite 5.4.6486 triple store cannot store very long literals (e.g., 20M chars of GML geographic data).
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 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. 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. 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 RDFPRO 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 RDFPRO invocation provides better performance, because input data is parsed once and I/O costs for accessing intermediate files are eliminated, as shown in Section 7.4.1 and 7.4.3. Task aggregation requires composition primitives and the support for reading and writing data at any point of the pipeline, two features of RDFPRO 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

---

122x 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

123Of 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.
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.
7.5 Using RDFPRO

RDFPRO binaries and public domain sources are available on its website. RDFPRO can be used in three ways: (i) as a command line tool able to process large datasets; (ii) as a web tool suited to smaller amounts of data uploaded/downloaded with the browser; and (iii) as a Java library embedded in applications. Users can extend RDFPRO via custom scripts and rule sets, while developers can create new processors by implementing a simple Java API and focusing on the specific task at hand, as efficient streaming, sorting, I/O, thread management, scripting, and composition facilities are already provided.

Examples of using RDFPRO as a command line and web tool are shown in Figures 29a and 29b where a pipeline is executed to compute the RDFS closure of some DBpedia data (70M triples) and return only rdfs:label triples of entities of type dbo:Company. The pipeline performs 6 tasks: (i) read data; (ii) compute RDFS closure using DBpedia TBox; (iii) keep rdf:type and rdfs:label quads; (iv) partition quads by subject, keeping partitions with object dbo:Company; (v) retain rdfs:label quads; (vi) write results. Besides the KnowledgeStore, an example of application using RDFPRO as an embedded Java library for RDF I/O, filtering, smushing, RDFS and rule-based inference, is PIKES [Corcoglioniti et al., to appear] (code publicly available); instructions for using the library are provided on the website.

A video showing the usage of RDFPRO is available on the website, together with a fully-working installation of the RDFPRO web interface, where users can try arbitrary commands and processing tasks.

---

124 http://rdfpro.fbk.eu/
125 Available on Maven Central: http://repo1.maven.org/maven2/eu/fbk/rdfpro/
126 http://pikes.fbk.eu
8 The ESO reasoner

<table>
<thead>
<tr>
<th>Changes wrt the KnowledgeStore Deliverable D6.2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>• reasoner updated to the new ESO version (Section 8.1);</td>
</tr>
<tr>
<td>• added reasoning via Contextualized Knowledge Repositories (Section 8.2);</td>
</tr>
</tbody>
</table>

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). There are rare cases where one can find a during-situation.

To describe these facts, the Event and Situation Ontology (ESO) has been developed. \[\text{Segers et al., 2015}\] 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:

- \((\#evID, \text{rdf:type, sem:Event})\)
- \((\#evID, \text{rdf:type, eso:LeavingAnOrganization})\)
- \((\#evID, \text{rdfs:label, sack})\)
- \((\#evID, \text{eso:employment-employer, dbp:Lincoln_City_F.C.})\)
• (#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)

Finally, ESO specifies the properties that can be used as predicate in assertions within a situation named graph. Two types of properties are used:

binary properties : these properties are modelled as object properties and they enable to relate two entities (e.g., see property “eso:employs” and “eso:notEmploys” in the situations instantiated for the firing event example previously considered);

unary properties : these properties are modelled as datatype properties and they enable to express facts such as that an entity exists or that some attribute has some relative value. Typically, the range of such properties is a boolean value type or a relative value.

Section 8.1 shows how the ESO reasoner works and describes its implementation. It can be downloaded from the GitHub repository\(^\text{127}\). An exhaustive description of classes and properties is available in Segers et al., 2015.

\(^{127}\)https://github.com/newsreader/eso
8.1 The tool

The tool is developed as a dedicated processor (@esoreasoner) of RDFpro (see Section 7), 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.

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 RDFpro 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).

- The event types associated to E are retrieved (for example, let E be a eso:LeavingAnOrganization 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). In the new version of the ESO ontology, an event can be attached to a time interval (let call it I), instead to a fixed instant: in this case the time attached to the situation is the interval ($-\infty$, t) or (t, $\infty$), where t is the initial/final time of I, depending whether we are instantiating a pre- or post-situation. For the during-case, the interval is linked to the situation as is.
  
  - 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

https://github.com/dkmfbk/eso-reasoner

NewsReader: ICT-316404 February 8, 2016
of triples (in the example, the objects of \textit{eso:hasSubject} and \textit{eso:hasObject} are \textit{eso:employment-employee} and \textit{eso:employment-employer}; let \( E \) have assertion for both of them).

* Let \( P \) be the object of the \textit{eso:hasProperty} assertion of the current \textit{eso:SituationAssertion} (for instance, it could be \textit{eso:isEmployedAt}).

* Some properties are considered mandatory for a particular event, but may be missing in the document (maybe because they are tacit or they were not extracted correctly). In this case, a placeholder URI is generated, and used as subject/object of the assertion.

* A new triple \( \langle a, p, b \rangle \) 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).

### 8.2 Reasoning with contexts

In this section, we propose to reinterpret and formalize the event model defined by the ESO ontology using a context-based framework for representation of Semantic Web data, the \textit{Contextualized Knowledge Repositories (CKR)} framework \cite{SerafiniHomola:2012,BozzatoSerafini:2012,BozzatoSerafini:2013}: this enables to exploit the structure and reasoning possibilities offered by the contextual framework in order to perform complex inferences about and inside knowledge associated with events. This can be seen as a logic-base alternative to the approach implemented in the ESO reasoner already described.

Intuitively, CKR is a description logics (DL) based framework defined as a two-layered structure: a lower layer contains a set of knowledge bases representing each context, while the upper layer contains context independent knowledge and meta-data defining the structure of contexts. The CKR framework has not only been presented as a theoretical framework, but also actual implementations based on its definitions \cite{BozzatoSerafini:2013,BozzatoSerafini:2014} have been proposed. In particular, in \cite{BozzatoSerafini:2013} we presented an implementation for the CKR framework over state-of-the-art tools for storage and inference over RDF graphs: intuitively, the CKR architecture can be implemented by representing the global context and the local object contexts as distinct RDF named graphs, while inference inside (and across) named graphs is implemented as SPARQL-based forward rules.

From a formal ontology point of view, in the approach we propose here we clearly distinguish \textit{ontology} from \textit{knowledge}. The upper layer of our system contains the underlying ontology, that is, the description of the organization of the world. This part lists types of entities, relations and constraints that are assumed to exist or to simply be possible. More generally, the upper layer should be thought as including two elements: a general (foundational or domain) ontology describing what can exist, and the organization of a set of knowledge modules characterizing roles and relationships in (typically social) standardized scenarios like economic transactions or soccer games. Regarding the first, we do...
not make a specific commitment towards one ontology or another although we explicitly commit to the existence of physical and social objects, like people and organizations, and temporal happenings like weddings and thunderstorms. These entities come with their usual physical and temporal properties: weight, shape, duration and so on. Regarding the modules, we do explore their role in the system to infer new facts from given knowledge and thus we will present their general setting and how they are used. The lower layer of the system, on the other hand, collects claims about what happens in the world, that is, claims about how things are or change at some time. Throughout the rest of this section these are what we call events, but note that they are not events in the ontological sense, rather they are descriptions of happenings.

Each event is associated with three state-like entities, namely happenings characterized by some continuously holding property or relation like “begin married”, “being running” or “being employed by” (e.g. see the notion of static perdurant in DOLCE [Masolo et al., 2003]). These state-like entities are called situations in our system; more specifically the situation holding before the event is called pre-situation, the one holding after post-situation and the one holding during the event itself is the during-situation. Informally, these situations are used to make explicit relevant properties that (a) persist during the whole event, or (b) hold (or don’t hold) before/after the event and their truth values change “because” of the event. We will use situations to formalize (and reason on) the preconditions for an event of a certain type to happen, the postconditions (or consequences) of its happening, and what can be taken as stable during the whole event of that type.

Although we take the received events at face value, it is possible that the news are imprecise or incomplete, that the text processing used to acquire the news is faulty and misinterprets them, that statements extracted from different sources about the same happenings contradict each other. For this reason, we take each piece of information extracted from an outside source as a contextual perspective on the world. This means that the content of a news is not equated to absolute knowledge (it is not indisputable). Indeed, the extracted information constitutes a context annotated with its original source (possibly with different degrees of reliability which can change over time). This allows to integrate pieces of information coming from different sources into large and articulated event descriptions by integrating different contexts (in turn, news statements). These extrapolated events reconstruct how an entity like a person or an organization, changes over time by changing its status (size, capacity, death) or its relationships with other entities (property acquisition, employment, marriage). Furthermore, missing information on these complex events can be detected via logical reasoning based on the ontology at the upper layer, leading to infer changes that have not been reported (or detected) in the news. Finally, note that, when a contradiction arises, the system can isolate the conflicting pieces of information and establish which contexts are to be kept apart and need to be verified.

8.2.1 Contextualized Knowledge Repositories

In the following we provide an informal summary of the definitions for the CKR framework: for a formal and detailed description and for complete examples, we refer to [Bozzato and ...]
A CKR is a two layered structure: (1) the upper layer consists of a knowledge base $\mathfrak{G}$, called global context, containing (a) meta-knowledge, i.e. the structure and properties of contexts, and (b) global (context-independent) object knowledge, i.e., knowledge that applies to every context; (2) the lower layer consists of a set of (local) contexts that contain locally valid facts and can refer to what holds in other contexts. The intuitive structure of a CKR knowledge base is depicted in Figure 30: in the following we detail its formal components and interpretation.

**Syntax.** The meta-knowledge of a CKR is expressed in a DL language containing the elements that define the contextual structure: the meta-vocabulary $\Gamma$ is a DL signature containing, in particular, the sets of symbols for context names $\mathbf{N}$, module names $\mathbf{M}$ and context classes $\mathbf{C}$, including the class of all contexts $\text{Ctx}$. Intuitively, modules represent pieces of knowledge specific to a context or a context class. The role $\text{mod}$ defined on $\mathbf{N} \times \mathbf{M}$ expresses associations between contexts and their modules. The meta-language $\mathcal{L}_\Gamma$ of a CKR is a DL language over $\Gamma$.

The knowledge in contexts of a CKR is expressed via a DL language $\mathcal{L}_\Sigma$, called object-language, based on an object-vocabulary $\Sigma$. The expressions of the object language are evaluated locally to each context, i.e., contexts can interpret each symbol independently. To access the interpretation of expressions inside a specific context or context class, we extend $\mathcal{L}_\Sigma$ to $\mathcal{L}_e\Sigma$ with eval expressions of the form $\text{eval}(X, C)$, where $X$ is a concept or role expression of $\mathcal{L}_\Sigma$ and $C$ is a concept expression of $\mathcal{L}_\Gamma$ (with $C \subseteq \text{Ctx}$). Intuitively, $\text{eval}(X, C)$ can be read as “the interpretation of $X$ in all the contexts of type $C$”.

We define a Contextualized Knowledge Repository (CKR) as a structure $\mathcal{R} = \langle \mathfrak{G}, \{K_m\}_{m \in \mathbf{M}} \rangle$ where: (i) $\mathfrak{G}$ is a DL knowledge base over $\mathcal{L}_\Gamma \cup \mathcal{L}_\Sigma$; (ii) every $K_m$ is a DL knowledge base over $\mathcal{L}_e\Sigma$, for each module name $m \in \mathbf{M}$. We note that the knowledge in a CKR can be expressed by means of any DL language: in our work, we consider $\text{SROIQ-RL}$ [Bozzato and Serafini, 2013] as language of reference. $\text{SROIQ-RL}$ is a restriction of $\text{SROIQ}$ syntax corresponding to OWL RL [Motik et al., 2009].
Semantics. The semantics of CKR basically extends the usual model-based semantics of DL knowledge bases to the two layered structure of the framework. A CKR interpretation is a structure \( \mathcal{I} = (\mathcal{M}, \mathcal{I}) \) s.t.: (i) \( \mathcal{M} \) is a DL interpretation of \( \Gamma \cup \Sigma \) (respecting the intuitive interpretation of \( \text{Ctx} \) as the class of all contexts); (ii) for every context \( x \in \text{Ctx}^\mathcal{M} \), \( \mathcal{I}(x) \) is a DL interpretation over \( \Sigma \) (with same domain and interpretation of individual names of \( \mathcal{M} \)). The interpretation of ordinary DL expressions on \( \mathcal{M} \) and \( \mathcal{I}(x) \) is as usual while \( \text{eval} \) expressions are interpreted as follows: for every \( x \in \text{Ctx}^\mathcal{M} \), \( \text{eval}(X, \mathcal{I}(x)) \) represents the union of all elements in \( X^\mathcal{I}(e) \) for all contexts \( e \) in \( \mathcal{C}^\mathcal{M} \).

A CKR interpretation \( \mathcal{I} \) is a CKR model of \( \mathcal{K} \) iff: (i) for \( \alpha \in L_\Sigma \cup L_T \) in \( \mathcal{K} \), \( \mathcal{M} \models \alpha \); (ii) for \( (x, y) \in \text{mod}^\mathcal{M} \) with \( y = m^\mathcal{M} \), \( \mathcal{I}(x) \models K_m \); (iii) for \( \alpha \in \mathcal{G} \cap L_\Sigma \) and \( x \in \text{Ctx}^\mathcal{M} \), \( \mathcal{I}(x) \models \alpha \). Intuitively, this means that \( \mathcal{I} \) verifies the contents of global and local modules associated with contexts and global object knowledge has to be propagated to local contexts.

Materialization calculus. Reasoning in CKR has been formalized as a materialization calculus [Krötzsch, 2010], a datalog-based calculus for instance checking in SROIQ-RL CKRs.

Intuitively, the calculus is based on a translation to datalog of the input CKR. It has three components: (i) the input translations \( I_{\text{glob}}, I_{\text{loc}}, I_{rl} \), where given an axiom \( \alpha \) and \( c \in \mathbb{N} \), each \( I(\alpha, c) \) is a set of datalog facts or rules encoding the contents of input global and local DL knowledge bases; (ii) the deduction rules \( P_{\text{loc}}, P_{rl} \), which are sets of datalog rules representing the inference rules for the instance-level reasoning over the translated axioms; and (iii) the output translation \( O \), where given an axiom \( \alpha \) and \( c \in \mathbb{N} \), \( O(\alpha, c) \) is a single datalog fact encoding the ABox assertion \( \alpha \) that we want to prove to be entailed by the input CKR (in the context \( c \)).

Intuitively, SROIQ-RL input \( I_{rl} \) and deduction \( P_{rl} \) rules provide the translation and interpretation of SROIQ-RL axioms from the input CKR. Global input rules in \( I_{glob} \) encode the interpretation of \( \text{Ctx} \) in the global context. Similarly, local input rules \( I_{loc} \) and deduction rules \( P_{loc} \) provide the translation and rules for the local \( \text{eval} \) expressions. The rules in \( O \) provide the translation of ABox assertions that can be verified to hold in a context \( c \) by applying the rules of the final program.

The translation of a CKR \( \mathcal{K} \) to its datalog program \( PK(\mathcal{K}) \) proceeds in four steps: we first translate \( \mathcal{G} \) in the global program \( PG(\mathcal{G}) \) by applying input rules \( I_{glob} \) and \( I_{rl} \) to \( \mathcal{G} \) and adding deduction rules \( P_{rl} \); then, for every context name \( c \in \mathbb{N} \) appearing in \( PG(\mathcal{G}) \), we compute its knowledge base \( K_c \) as the set of modules \( K_m \in \mathcal{K} \) s.t. \( \text{mod}(c, m) \) is proved by \( PG(\mathcal{G}) \); we translate each local program \( PC(c) \) by applying input rules \( I_{loc} \) and \( I_{rl} \) to \( K_c \) and adding deduction rules \( P_{loc} \) and \( P_{rl} \); the final CKR program \( PK(\mathcal{K}) \) is then obtained as the union of \( PG(\mathcal{G}) \) with all local programs \( PC(c) \). We say that \( \mathcal{K} \) entails an axiom \( \alpha \) in a context \( c \in \mathbb{N} \) if \( PK(\mathcal{K}) \models O(\alpha, c) \). We can show (see [Bozzato and Serafini, 2013]) that the presented rules and translation process provide a sound and complete calculus for instance checking in SROIQ-RL CKR.

CKR implementation on RDF. We recently presented a prototype [Bozzato and Serafini, 2013] that implements the forward reasoning procedure over CKR defined by the materialization calculus. The prototype accepts RDF input data expressing OWL-RL axioms.
and assertions for global and local knowledge modules: these different pieces of knowledge are represented as distinct named graphs, while we encoded in a OWL vocabulary the CKR contextual primitives (e.g. the class Context of all context individuals, the class Module of all modules and the property hasModule corresponding to the role mod). The prototype is based on an extension of the Sesame RDF Framework and structured in a client-server architecture: the main component, called CKR core and residing in the server-side part, offers the ability to compute and materialize the inference closure of the input CKR, add and remove knowledge and execute queries over the complete CKR structure.

The distribution of knowledge in different named graphs asks for a component to compute inference over multiple graphs in a RDF store, since inference mechanisms in current stores usually ignore the graph part. This component has been realized as a general software layer called SPRINGLES (SParql-based Rule Inference over Named Graphs Layer Extending Sesame) \cite{bozzato2013springles}. Intuitively, SPRINGLES provides methods to demand a closure materialization on the RDF store data: rules are encoded as (named graphs aware) SPARQL queries and it is possible to customize both the used ruleset and the evaluation strategy.

In our case, the ruleset basically encodes the rules of the presented materialization calculus. The rules are evaluated with a strategy that follows the same steps of the translation process defined for the calculus. The plan goes as follows: (i) we compute the inference closure on the graph for global context $G$, by a fixpoint on rules corresponding to $P_{rl}$; (ii) we derive associations between contexts and their modules, by adding dependencies for every assertion of the kind $\text{hasModule}(c, m)$ in the global closure; (iii) we compute the closure of the contexts, by applying rules encoded from $P_{rl}$ and $P_{loc}$ and resolving $\text{eval}$ expressions by the metaknowledge information in the global closure.

### 8.2.2 Representing events in CKR: CKR-ESO ontology

We can now describe how we translated and implemented a first prototype of the ESO model in the form of a contextualized ontology for the CKR, that we call the CKR-ESO ontology.

In this model, the event and situation structures are modelled in the metaknowledge. Similarly to the ESO model, each event instance is associated in the metaknowledge with its pre-, during- and post-situations using the object properties $\text{hasPreSituation}$, $\text{hasPostSituation}$ and $\text{hasDuringSituation}$, subproperties of $\text{hasSituation}$. Situation elements associated with events can be generated automatically by SPRINGLES rules when importing an event.

Each event is represented in the metaknowledge as an instance of the class Event: in particular, each event is associated, analogously to the ESO model, with a subclass of the Event class that determines the type of associated situations: in particular, DynamicEvents (e.g. ChangeOfPossession, Constructing) are typically characterized by their pre- and post-situations, while StaticEvents (e.g. BeingOperational) by their during-situations.

\url{http://www.openrdf.org/}
This classification is provided by restrictions over the definition of such classes. For example, for the ChangeOfPossession event class, the CKR-ESO ontology states that:

\[
\text{ChangeOfPossession} \subseteq \forall \text{hasPreSituation}.\text{Pre}_\text{ChangeOfPossession} \\
\text{ChangeOfPossession} \subseteq \forall \text{hasPostSituation}.\text{Post}_\text{ChangeOfPossession}
\]

Each event individual is associated with a knowledge module that corresponds to the RDF graph of the event in the ESO model. This association is represented in the metaknowledge by the property \text{hasEventModule}. The following chain axiom is defined over this property, asserting that situations related to an event inherit the facts asserted in the event module:

\[(\text{hasSituation})^{-} \circ \text{hasEventModule} \subseteq \text{hasModule}.\]

As defined by the ESO model, we expect to find in the event module the instantiation for all the required roles involved in the event.

The class \text{Situation} is defined as a subclass of the \text{Context} class in the CKR vocabulary: in other words, in our model we consider situations and their local knowledge as contexts. The particular (pre, post and during) situations associated with event types are modelled by specific context classes. Thus, for example, we have that the pre- and post-situations for events of type ChangeOfPossession are classified as members of the classes \text{Pre}_\text{ChangeOfPossession} and \text{Post}_\text{ChangeOfPossession}. The association between such type of situations and their local axioms (i.e. what its modelled in the ESO ontology by situation assertions) is performed by linking specific knowledge modules to these context classes. For example, in CKR-ESO we declare that every pre-situation of ChangeOfPossession is associated with the knowledge module \text{pre}_\text{change-of-posses-m} and post-situations to \text{post}_\text{change-of-posses-m}:

\[
\text{Pre}_\text{ChangeOfPossession} \subseteq \exists \text{hasModule}.\{\text{pre}_\text{change-of-posses-m}\} \\
\text{Post}_\text{ChangeOfPossession} \subseteq \exists \text{hasModule}.\{\text{post}_\text{change-of-posses-m}\}
\]

Situation assertions are thus encoded inside these specific modules: the assertions can be basically translated to chain axioms across the roles specified in the event. For example, assertions for pre-situations of ChangeOfPossession stating that:

\[
\text{hasInPossession}(\text{possession-owner}_1,\text{possession-theme}) \\
\text{notHasInPossession}(\text{possession-owner}_2,\text{possession-theme})
\]

is translated in CKR-ESO to these chain axioms across role properties:

\[
(\text{possession-owner}_1)\neg \circ \text{possession-theme} \subseteq \text{hasInPossession} \\
(\text{possession-owner}_2)\neg \circ \text{possession-theme} \subseteq \text{notHasInPossession}
\]

We now can show how to represent our example event from Section 8 using the CKR-ESO model: we depict this modelling in Figure 31. In the global context, we define event1 to be of type ChangeOfPossession and associate it with its situations:

\[
\text{ChangeOfPossession}(\text{event1}) \\
\text{hasPreSituation}(\text{event1}, \text{pre-event1}) \\
\text{hasPostSituation}(\text{event1}, \text{post-event1})
\]
By the above axioms for such event type, we know that the pre- and post-situations of \textit{event1} have to be of type \textit{Pre}\_ChangeOfPossession and \textit{Post}\_ChangeOfPossession. By metalevel reasoning, this implies that:

\[
\text{hasModule}(\text{pre-event1, pre-change-of-possession-m}) \\
\text{hasModule}(\text{post-event1, post-change-of-possession-m})
\]

and thus the situation assertions associated with the pre- and post-situations of \textit{ChangeOfPossession} are imported in the two situations\footnote{In Figure \ref{fig:example-event}, we abbreviate classes of pre- and post-situations of \textit{ChangeOfPossession} with \textit{Pre}\_CoP and \textit{Post}\_CoP and their modules with \textit{pre}\_cop-m and \textit{post}\_cop-m.}. Moreover, the graph associated with the original event is now defined as a module associated with \textit{event1} in the metalevel: \text{hasEventModule}(\textit{event1, event1}_m).

The \textit{event1}_m module (i.e. the associated RDF graph) now contains the following facts that are shared with all the situations associated with this event:

\[
\begin{align*}
\text{possession-owner}_1(\textit{event1}, \text{Ford}) \\
\text{possession-owner}_2(\textit{event1}, \text{Tata}\_\text{Group}) \\
\text{possession-theme}(\textit{event1}, \text{Jaguar and Land Rover})
\end{align*}
\]

Using the situation assertions in the module associated with the pre-situation, the CKR thus derives the following facts in the context of \textit{pre-event1}:

\[
\begin{align*}
\text{hasInPossession}(\text{Ford, Jaguar and Land Rover}) \\
\text{notHasInPossession}(\text{Tata}_\text{Group, Jaguar and Land Rover})
\end{align*}
\]

Similarly, in the context of \textit{post-event1} we obtain:
We note that representation of the described event can be completed with its associated during-situation: among the facts that are known to hold during the event, for example, we can assert the existence of the actors and of the possession theme (using the exists property in the ESO).

This contextual re-interpretation of the ESO model can bring several advantages from the point of view of reasoning capabilities inside and across events. First of all, every aspect of the reasoning procedure is now strictly ruled by logical reasoning: situation assertions and their association with the type of situation are now directly modelled by the CKR structure and local axioms, without demanding an external reasoner to consider the situation rules and the local reasoning inside situations. Furthermore, the propagation of global object knowledge to local knowledge allows the use of context independent background knowledge in local reasoning. In our example, we can assert in the global knowledge that both actors (Ford and Tata Group) are classified as car companies and their features can be used in local reasoning. More in general, the advantages of an explicit and structured representation of contexts (as the one offered by the CKR) with respect to a modelling based on reification have been shown in [Bozzato et al., 2013].

On the other hand, the clear separation of meta and object level reasoning can be exploited to exchange information across the two levels. For example, depending on the type and specific patterns of situation and events, by adding custom SPRINGLES rules it is possible to generate implicit events that have to occur for the completion of the event sequence in a story. In our example, if we have a second event event2 representing another ChangeOfPossession of Jaguar and Land Rover between two companies Company1 and Company2, different than the two companies from event1, and event2 has a timestamp greater than event1, then we can infer that there have been another two events (possibly being the same one): in one Jaguar and Land Rover has been sold from Tata Group and in the other it has been acquired from Company1. Similarly, we can recognize cases in which we can assert the equality of certain situations: this can be used to compile sequences of events in a story.

Metalevel information for situations and events can be derived from local reasoning: we might recognize incompatible descriptions of the same event from different news. For example, let us suppose a different representation of the scenario shown in the example in Figure 31: assume that event1 is now classified as Buying (subclass of ChangeOfPossession) while another event event2 is extracted as Selling (also subclass of ChangeOfPossession), but they both represent the same conceptual event (i.e. the acquisition of Jaguar and Land Rover by Tata Group from Ford). Thus, at the level of the metaknowledge, the two events are modelled as:
Buying(event1)
hasPreSituation(event1, pre-event1)
hasPostSituation(event1, post-event1)

Selling(event2)
hasPreSituation(event2, pre-event2)
hasPostSituation(event2, post-event2)

Since they represent the same happening, the event modules event1_m and event2_m basically share the same contents: that is, the actors are the same and they take the same role. However, suppose that, due to the extraction from different news sources, the metamodel property sem:hasTime associated to post-event1 has value “August 26th, 2007” while the value associated to pre-event2 is “August 28th, 2007”. Then, using this metalevel information and the local contents of the event modules, we can easily write a reasoning rule that recognizes that the two events are incompatible and adds the assertion event1 incompatibleWith event2 in the global context. Similarly, we can recognize inconsistent situations by local reasoning: this can be used both to exclude further inferences from inconsistent contexts, by marking as “inconsistent” the situation individual in the meta-knowledge, but also to repair (possibly with some ad-hoc rules) the local axioms. We note that, on the other hand, this kind of reasoning requires to define ad-hoc rules to recognize such different situations.

Another interesting possibility is the one of having inter-situation knowledge propagation. For example, if two situations or two events are recognized as consequent in a story, unmodified knowledge from the previous situations can be propagated to subsequent situations (e.g. the marital status of Obama did not change when he was elected US president). This clearly presents problems of non-monotonicity, since one has to consider which knowledge can be seamlessly propagated without incurring in contradictory states. In this regard, we recently introduced in CKR a notion of defeasible axioms and their overriding across different contexts [Bozzato et al., 2014].

We are still investigating on the scalability of this approach to millions of documents (and, hence, billions of triples). For the time of the NewsReader project, we will use the rule-based ESO reasoner described in Section 8.1 to perform the materialization of Situations and implied assertions.
9 Related Work

The development of frameworks able to store integrated and interlinked unstructured and structured content has not been deeply explored in the literature, although some relevant works closely related to our contribution do exist: the KIM Platform (Ontotext), Apache Stanbol, and the Linked Media Framework.

The KIM Platform [Popov et al., 2003] aims at providing a platform for semantic annotations of documents, focusing on named entity recognition and linking to a knowledge base of known entities. The platform’s main components are a document index, a knowledge base and an annotation pipeline. The document index, based on Lucene stores documents with their metadata and the entities recognized within them. The knowledge base contains the RDFS description of ~80K entities of international relevance (background knowledge) as well as entities extracted from documents, based on a specifically-designed ontology (KIMO) defining ~150 top-level entity classes and associated properties. The annotation pipeline is based on the Gate NLP suite extended to leverage information in the knowledge base, and allows the automatic annotation of documents with the entities they contain, typed with respect to KIMO and linked to known entities in the knowledge base. Several APIs and UIs are provided for document storage and annotation as well as for retrieving entities and documents using queries combining keywords and entities and allowing the navigation from documents to referenced entities and back. KIM has been used in production at several news providers such as BBC, more recently adopting the PROTON upper ontology in place of KIMO and selected LOD data as background knowledge. The methodology and the software architecture for these applications are described in Georgiev et al., 2013. Compared to our approach, the information extraction pipeline in KIM is fixed and closely tied to a specific ontological schema for entities (KIMO, then PROTON), whereas the KnowledgeStore is agnostic with respect to which pipeline, ontologies and background knowledge are used.

Apache Stanbol [Gönül and Sinaci, 2012], originated in the IKS Project, is a modular server exposing a configurable set of ReST services for the enhancement of unstructured textual contents. Stanbol main goal is to complement existing CMSs with semantic annotation, indexing and retrieval functionalities. CMS documents and their metadata are fed to the Stanbol server, where a pipeline of content enhancers is applied to extract entities

---

9.1 Changes wrt the KnowledgeStore Deliverable D6.2.2

- brand new Section, added to address the request of the reviewers to highlight the differences wrt to other state-of-the-art approaches;

---

[133] https://gate.ac.uk/
[134] proton.semanticweb.org
[135] https://stanbol.apache.org/
and additional metadata (e.g., language, topics). Extracted data are augmented with LOD data, and the result is indexed inside Stanbol in a triplestore (similar to the KnowledgeStore) as well as in a SOLR\(^\text{137}\) full-text index, supporting respectively SPARQL queries and keyword search. While the KnowledgeStore provides a scalable and reliable primary storage for resources, Stanbol is mainly focused on their indexing for search purposes, and thus their main storage remains in external CMSs.

The Linked Media Framework (LMF, \cite{Kurz et al., 2014}\(^\text{138}\)) offers storage and retrieval functionalities for multimedia contents annotated with LOD data. Annotations are provided by external content enhancers such as Stanbol, while the focus of the LMF is on storage and retrieval services as in the KnowledgeStore. Similarly to Stanbol, the LMF data server is based on a triplestore (Sesame) storing annotations as RDF triples and on a SOLR full-text index storing document texts as well as selected metadata and annotation values chosen via XPath-like LDPath expressions\(^\text{139}\); the two storages enable respectively SPARQL queries and keyword-based document search. Similarly to the KnowledgeStore, a ReST API extending the Linked Data HTTP publishing scheme allows read/write access to stored contents.

Compared to the KnowledgeStore, KIM, Stanbol and LMF all adopt a ‘two-layer’ model consisting only of resources (text and metadata indexed in a full-text index) and entities (triples indexed in a triplestore). Indeed, storing and querying mention attributes is not a goal of these frameworks. Although mention data could be stored as additional attributes of resources and/or entities, this is not the intended use of these layers and this expedient may lead to inefficiencies or it may be not feasible at all\(^\text{140}\). On the other hand, using the KnowledgeStore as a two-layer system is possible too, but with a small overhead imposed by the unused Mention layer. Therefore, a fair quantitative comparison between the KnowledgeStore and these frameworks is not possible, as they provide different feature sets and they target different usage scenarios. Beyond the different number of layers, another distinctive feature of the KnowledgeStore compared to KIM, Stanbol and LMF is its use of named graph to track the provenance of entities and axioms and to qualify the context where a particular axiom holds.

Apart the mentioned works, some investigations were carried out on document repositories based on semantics (e.g., \cite{Bang and Eriksson, 2006, Eriksson, 2007}). In these approaches, ontologies encode the domain vocabulary and the document structure, and they are used for annotating documents and document parts. However, the repositories adopting these approaches: (i) emphasize the document structure (e.g., tables, title) rather than document content, (ii) they do not foresee an integrated framework for storing semantic content and unstructured documents together, and (iii) they are not meant to be applied in big data contexts.

Relevant for our work is also the contribution presented in \cite{Croset et al., 2010}. The

\(^{137}\)http://lucene.apache.org/solr/

\(^{138}\)https://code.google.com/p/lfm/

\(^{139}\)https://code.google.com/p/ldpath/

\(^{140}\)For instance, storing mentions as Entity data in the triplestore may lead to an ‘explosion’ of its size, as noted in Section 4.
authors present a framework, based on a RDF triplestore, that enables querying the bioinformatics scientific literature and structured resources at the same time, for evidence of genetic causes, such as drug targets and disease involvement. Differently from our approach, this work does not support storing unstructured content (triplestores currently provide only a limited support for integrating knowledge with unstructured resources, often consisting in simple full text search capabilities on RDF literals), and the framework is focused only on specific types of named entities appearing in the unstructured content, whereas a rich, unconstrained set of entities and mentions can be managed in the KnowledgeStore. Another relevant work, in the biomedical domain, is Semantic Medline\textsuperscript{141} a web application that summarizes MEDLINE citations returned by a PubMed search. Natural language processing is performed to extract semantic predications (the equivalent of entity axioms in KnowledgeStore terminology) from titles and abstracts. However, differently from the KnowledgeStore, Semantic Medline has a fixed domain-specific data model, built tailored on that application, and predications can be effectively navigated only on a reasonably small selection of citations (max 500 on the web site) with no possibility to perform structured queries on the whole corpus (to this respect, a global index of predications seems missing). Furthermore, while capable of handling large quantity of resources (21M Medline citations, see Jonnalagadda et al., 2012\textsuperscript{141}), the semantic content extracted and to be handled is proportionally rather small (~57.6M predications of 26 types; cf. with Cars (Ver. 2) KnowledgeStore instance, with 535M triples from 1.3M news articles).

Although exploited in a different context, dealing with much smaller quantity of content, also semantic desktop applications such as MOSE [Xiao and Cruz, 2006\textsuperscript{142}] and Nepomuk\textsuperscript{142} are partly related with the contribution here presented. Semantic desktop applications enrich documents archived on the personal PC of a user with annotations coming from ontologies. However, annotations are attached to the object associated to the document, and not to its content, thus not fully supporting the interlinking between unstructured and structured content.

\textsuperscript{142}http://nepomuk.semanticdesktop.org/
Conclusions

In this deliverable we documented the final 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.

This version of the NewsReader KnowledgeStore provides the core infrastructure functionalities and several improvements have been made compared to the previous releases: the implementation of the user interface (Section 3.4) and additional methods to navigate content stored in different layers of the KnowledgeStore; some updates on the population (including several different case studies, showing the scalability of the KnowledgeStore, see Section 5); the description of some new features in the KnowledgeStore architecture (e.g., ElasticSearch back-end); the development of a tool for processing large RDF datasets (RDFpro, see Section 7); the new ESO reasoner (Section 8). To assess the KnowledgeStore scalability, we reported (Section 6) on a thorough evaluation of the its performances, covering both data population and data retrieval with different dataset sizes and numbers of concurrent clients.

The contributions presented in this deliverable cover the three-year activities carried out in WP6 (KnowledgeStore), and led to the following publications (in chronological order):


- RDFpro: an Extensible Tool for Building Stream-Oriented RDF Processing Pipelines (Francesco Corcoglioniti, Marco Rospocher, Marco Amadori, Michele Mostarda), In


Demonstrating the Power of Streaming and Sorting for Non-distributed RDF Processing: RDFpro (Francesco Corcoglioniti, Alessio Palmero Aprosio, Marco Rospocher), In ISWC 2015 Posters & Demonstrations Track, within the 14th International Semantic Web Conference (ISWC 2015), Bethlehem, USA, October 11-15, 2015, 2015.

For future developments of the KnowledgeStore, beyond the scope of the NewsReader project, we identified several research challenges worth investigating.

Concerning the reasoning services, one possibility is to investigate how to extend 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, it is possible 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, one extension to investigate concerns the possibility of offering additional query mechanisms, based on indexing technologies such as Apache Lucene / Solr / Elastic Search, enabling support for full text search on the documents stored into the KnowledgeStore. The implementation of the ElasticSearch back-end for the KnowledgeStore, described in Section 4.1.2, goes already in this direction.
To improve the user experience in exploring and navigating the KnowledgeStore content, the KnowledgeStore UI may be extended to offer the possibility of integrating techniques based on faceted searching\textsuperscript{143} and browsing\textsuperscript{144} on top of the KnowledgeStore.

\textsuperscript{143}http://en.wikipedia.org/wiki/Faceted_search
\textsuperscript{144}http://www.sindicetech.com/pivotbrowser.html
References


NewsReader: ICT-316404 February 8, 2016


