Knowledge store design

Deliverable D6.1

Version FINAL

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

Affiliation: (1) FBK

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

ICT 316404

Grant Agreement No.	316404
Project Acronym	NEWSREADER
Project Full Title	Building structured event indexes of large volumes of financial and economic data for decision making.
Funding Scheme	FP7-ICT-2011-8
Project Website	http://www.newsreader-project.eu/
Project Coordinator	Prof. dr. Piek T.J.M. Vossen VU University Amsterdam Tel. + 31 (0) 20 5986466 Fax. + 31 (0) 20 5986500 Email: piek.vossen@vu.nl
Document Number	Deliverable D6.1
Status & Version	FINAL
Contractual Date of Delivery	June 2013
Actual Date of Delivery	July 16, 2013
Type	Report
Security (distribution level)	Public
Number of Pages	61
WP Contributing to the Deliverable	WP6
WP Responsible	FBK
EC Project Officer	Sophie Reig

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

Affiliation: (1) FBK

Keywords: knowledge store, unstructured content, mentions, entities

Abstract: Despite the widespread diffusion of structured data sources and the public acclaim of the Linked Open Data initiative, a preponderant amount of information remains nowadays available only in unstructured form, both on the Web and within organizations. While different in form, structured and unstructured contents speak about the very same entities of the world, their properties and relations; still, frameworks for their seamless integration are lacking. In this deliverable we describe the design of the 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.

Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.1	30 April 2013	Draft of Deliverable Skeleton	Marco Rospocher	
0.2	9 May 2013	Refinement of the Deliverable Skeleton	Francesco Corcoglioniti	
0.3	20 May 2013	First Draft of Introduction	Marco Rospocher	1
0.4	22 May 2013	First Draft of Related Work, Architecture, Data Model	Marco Rospocher, Roldano Cattoni, Francesco Corcoglioniti	2,4,5
0.5	23 May 2013	Revision of drafted parts	Marco Rospocher, Roldano Cattoni, Francesco Corcoglioniti, Bernardo Magnini, Luciano Serafini	1,2,4,5
0.6	28 May 2013	Added first draft of Interfaces	Marco Rospocher	3
0.7	28 May 2013	Revision of Interfaces	Roldano Cattoni, Francesco Corcoglioniti	3
0.8	03 June 2013	Revision of Interfaces, Architecture, Data Model. Draft of Appendixes	Francesco Corcoglioniti	2,3,4,A,B,C
0.9	03 June 2013	Revision of Interfaces, Architecture	Roldano Cattoni, Bernardo Magnini, Luciano Serafini	3,4
1.0	11 June 2013	Revision of Related Work, Interfaces, Architecture,	Marco Rospocher	3,4,5
1.1	14 June 2013	Revision of Data Model, Interfaces - Added Section 5.2	Francesco Corcoglioniti	2,3,5.2
1.2	17 June 2013	Revision of whole document - Sent for internal review	Marco Rospocher	All
1.3	01 July 2013	Revision of Data Model, and appendixes	Francesco Corcoglioniti	2,A,B,C
1.4	01 July 2013	Revision of Data Model, and appendixes	Marco Rospocher	All
	07 July 2013	Internal Review of Section 1 and 2	German Rigau	1,2
1.5	08 July 2013	Revision according to the internal review comments	Marco Rospocher	1,2
	13 July 2013	Internal Review of Section 3, 4 and 5	German Rigau	3, 4, 5
1.6	16 July 2013	Revision according to the internal review comments	Francesco Corcoglioniti, Roldano Cattoni	3, 4, 5

Executive Summary

This document presents the design 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) of Work Package WP6 (KnowledgeStore).

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

We detail the design aspects 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 detail the internal component organization of the KnowledgeStore (the KnowledgeStore *architecture*), discussing the technological choices we made.

We comment the position the KnowledgeStore with respect to other state of the art contributions for the integrated and interlinked storage of unstructured and structured content, and we conclude with an overview of the KnowledgeStore implementation plan and some ideas for evaluating the KnowledgeStore contribution.

The contributions here presented will also appear in the proceedings of the 7th IEEE International Conference on Semantic Computing (ICSC2013) [Rospocher *et al.*, 2013].

Contents

List of Figures

1 Introduction

This document describes the KnowledgeStore, the infrastructure that will be used in NewsReader to store, retrieve, and reason about the knowledge extracted from financial and economical news.

1.1 The KnowledgeStore Vision

The rate of growth of digital data and information is nowadays continuously increasing. While the recent advances in Semantic Web Technologies (e.g., the Linked Data[1] initiative), have favoured the release of large amount of data and information in structured machine-processable form (e.g., RDF dataset repositories), a huge amount of content is still available and distributed through websites, company internal Content Management System (CMS) and repositories, in an unstructured form, for instance as textual document, web pages, and multimedia material (e.g., photos, diagrams, videos). Indeed, as observed in [Gantz and Reinsel, 2011], unstructured data accounts for more than 90% of the digital universe.

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

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

In this document we describe the design of the KnowledgeStore, a framework that contributes to bridge the unstructured and structured worlds, enabling to jointly store, manage, retrieve, and semantically query, both typologies of contents. Figure 1 shows schematically how the KnowledgeStore manages these contents in its three *representation layers*. On the one hand (and similarly to a file system) the *resource layer* stores unstructured content in the form of resources (e.g., news articles, multimedia files), each having a textual or binary representation and some descriptive metadata. Information stored in this level is typically noisy, ambiguous, and redundant, with the same piece of information potentially represented in different ways in multiple resources. On the other hand, the *entity layer* is the home of structured content, that, based on Knowledge Representation and Semantic

[1] `http://linkeddata.org`

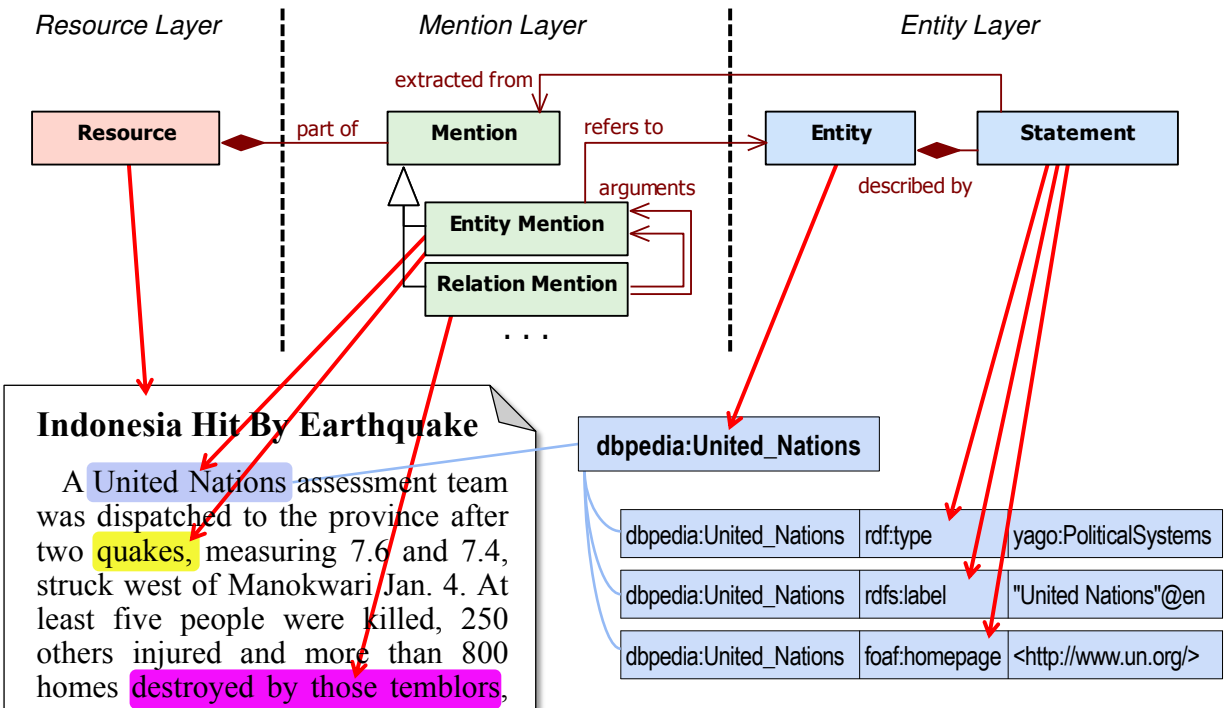

Figure 1: KnowledgeStore Content.

Web best practices, consists of ⟨subject, predicate, object⟩ *statements*, which describe the *entities* of the world (e.g., persons, locations, events), and for which additional metadata is kept to track their provenance and to denote the formal *contexts* where they hold (e.g., in terms of time, space, point of view). Differently from the resource layer, the entity layer aims at providing a formal and concise representation of the world, abstracting from the many ways it can be encoded in natural language or in multimedia, and thus allowing the use of automated reasoning to derive new statements from asserted ones [De Bruijn and Heymans, 2007]. Between the aforementioned two layers is the *mention layer*. It indexes *mentions*, i.e., snippets of resources (e.g., some characters in a text document, some pixels in an image) that denote something of interest, such as a an entity or a statement 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 exploit-

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

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

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

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

The idea behind the KnowledgeStore was preliminary investigated in [Cattoni *et al.*, 2012] and tested in the scope of the LiveMemories project[2]. However, we highly revised the design of the previous version, introducing significant enhancements: the new version of the KnowledgeStore, currently under implementation, 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 (potentially, billions of documents versus a few hundreds of thousands), and (iii) a semantic query mechanism over its content (no reasoning services was previously offered).

[2] `http://www.livememories.org/`

1.2 Role of the KnowledgeStore in NewsReader

The goal of the NewsReader Project[3] is to process daily economical and financial news in order to extract events (i.e., what happened to whom, when and where – e.g., "The Black Tuesday, on 24th of October 1929, when United States stock market lost 11% of its value"), and to organize these events in coherent narrative stories, combining new events with past events and background information. These stories are then offered to professional decision-makers, that by means of visual interfaces and interaction mechanisms will be able to explore them, exploiting their explanatory power and their systematic structural implications, to make well-informed decisions. Achieving these challenging goals requires:

- to process document resources, detecting mentions of events, event participants (e.g., persons, organizations), locations, time expressions, and so on;
- to link extracted mentions with entities, either previously extracted or available in some structured domain source, and coreferring mentions of the same entity;
- to complete entity descriptions by complementing extracted mention information with available structured knowledge (e.g., DBPedia[4], corporate databases);
- to interrelate entities (events and their participants, in particular) to support the construction of narrative stories;
- to reason over events to check consistency, completeness, factuality and relevance;
- to store all this huge quantity of information (on resources, mentions, entities) in a scalable way, enabling efficient retrieval and intelligent queries;
- to effectively offer narrative stories to decision makers.

A framework like the KnowledgeStore can effectively contribute to address such kind of requirements[5].

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

- the *resource layer* stores the unstructured financial news and their annotations;
- the *mention layer* identifies fragments of news denoting entities (e.g., a take-over event), relation between entity mentions (e.g., event participation), numerical quantities (e.g., a share price);
- the *entity layer*[6] stores the structured descriptions of those entities extracted from resources and merged with available structured knowledge (e.g., Linked Data sources, corporate databases).

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

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

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

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

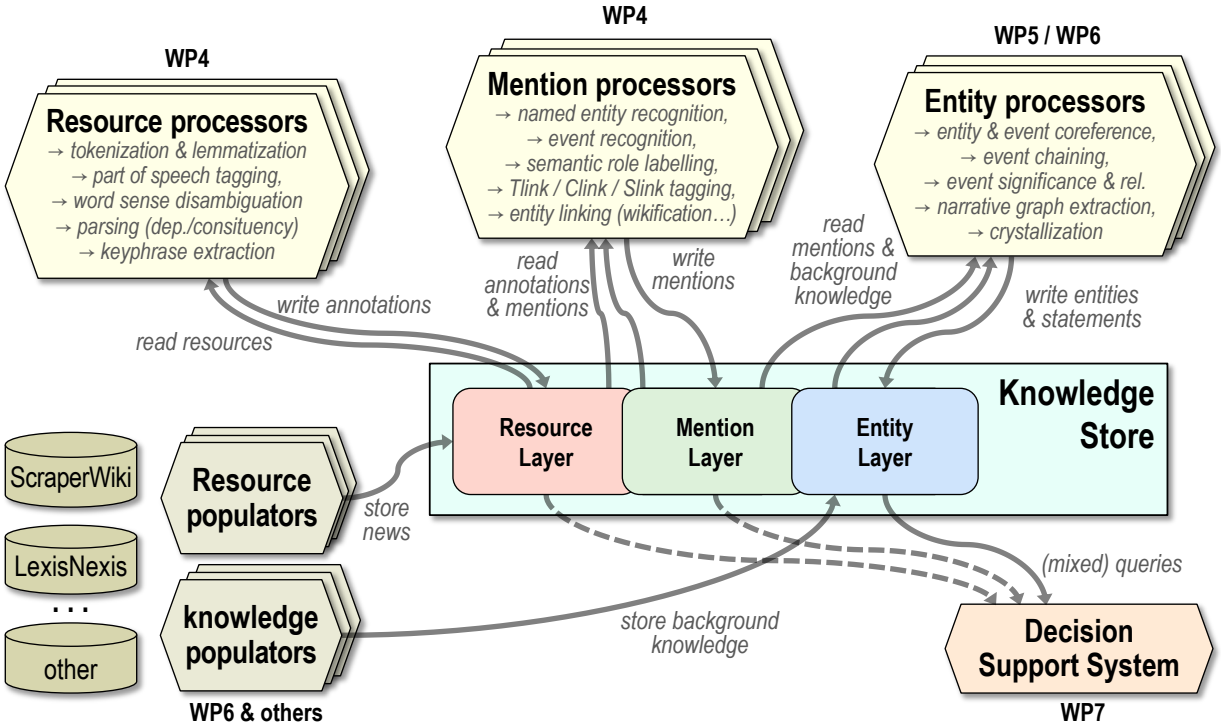

Figure 2: The role of the KnowledgeStore in NewsReader.

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

- *Resource* and *knowledge populators*. These modules enable the bulk loading of structured and unstructured contents in the KnowledgeStore.

- *Resource Processors*. These modules, as part of WP4 activities, work at the resource layer, and take care of performing a pre-processing of the text document, enriching it with linguistic annotations.

- *Mention Processors*. These modules, as part of WP4 activities, work at the resource and mention layers, exploiting the results of resource processors to instantiate mentions via named entity and event recognition, semantic role labelling, and so on.

- *Entity Processors*. These modules, as part of WP5 and WP6 activities, work at the mention and entity layers, exploiting the results of mention processors to instantiate, link, or enrich entities performing tasks such as coreference and knowledge fusion.

- *Decision Support System*. Finally, as part of WP7 activities, the decision support system 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 focuses on providing efficient retrieve/query mechanisms, while a basic implementation of the CRUD operations is offered, allowing the external modules to have full access (and control) on the content of the KnowledgeStore[7].

NewsReader will be tested on 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.

1.3 Content of this Deliverable

The deliverable is organized as follows. In Section 2 we present in details the KnowledgeStore data model, highlighting both its configurable, cross-domain design and its current configuration for NewsReader. In Section 3 we illustrate how the other NewsReader modules can interact with the KnowledgeStore, detailing in particular the type of (semantic) requests they can submit to it, while in Section 4 we describe the KnowledgeStore architecture, presenting each module composing the framework and the physical implementation of the data model. In Section 5 we briefly overview related state-of-the-art approaches and technologies. Section 6 concludes with some final remarks.

Before proceeding, we want to remark that this deliverable presents a concrete, yet initial, proposal for the data model, interfaces and architecture of the KnowledgeStore. This because several activities tightly related to the KnowledgeStore are going on at the time of writing this deliverable, and their outcomes is foreseen at M6 (D1.3: Application and system requirements - draft; D3.1: Annotation module; D4.1: Resources and linguistic processors) or later (M9, D4.2.1: Event Detection – version 1; M12, D2.1: System Design - draft). Therefore, some revision of the design choices here presented may occur, and, if the case, will be documented in the next WP6 Deliverables (D6.2.1: KnowledgeStore – version 1; D6.2.2: KnowledgeStore – version 2; D6.2.3: KnowledgeStore – version 3).

[7]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 has to be investigated. Therefore the KnowledgeStore does not handle them, although it offers to each module the basic operations to implement the more appropriate strategy to cope with them.

2 The KnowledgeStore Data Model

The data model defines what information can be stored in the KnowledgeStore, in accordance with the modelling and specification work of WP3, WP4, WP5 and, in particular, the annotation format (its format and structure will be described in Deliverable D2.1: System Design - draft, while its content is documented in Deliverable D3.1: Annotation Module). Therefore, it serves both as a basis for the design of the KnowledgeStore architecture and interfaces, and as a shared model that permits linguistic processors, other processing components and the decision support tool suite to cooperate.

The dependencies to ongoing design activities in other WPs, the need to accommodate possible changes during the project lifetime, and the envisioned support to a broad range of applications, make flexibility a key requirement for the KnowledgeStore data model. This is addressed through the design of a minimalist, configurable data model, centred around the key concepts of resource, mention and entity described by statements within a context. The data model is then configured (and re-configured) for use in NewsReader 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 OWL 2 formal specifications are documented in Appendices B and C. The presentation is at a conceptual level with no implication on the physical organization of data, which is addressed as part of Section 4.2.

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, described next and highlighted in the figure. Resources and mentions are described using a closed but configurable set of types, attributes and relations, while entities are described with an open set of statements annotated with metadata attributes (e.g., for provenance) and holding inside specific contexts. Resources, mentions and entities are identified by URIs, assigned by the system in the first two cases, and externally for entities. Statements are identified by their $\langle$*subject*, *predicate*, *object*, *context*$\rangle$ components, while contexts are identified both by a set of externally-assigned contextual metadata (e.g., for time, space, point of view) and by a system-generated URI. By design, types, attributes and relations are denoted with URIs (hereafter abbreviated using qualified names and a KnowledgeStore default namespace[8]) and the model is assimilable to an OWL 2 ontology [Motik *et al.*, 2009]. This allows to encode both the model definition and its instance data using RDF [Beckett, 2004] (e.g., for interfacing with the KnowledgeStore or for Linked Data publishing), and to use other Semantic Web technologies to deal with represented data; in particular, instance data at the entity layer can be encoded using RDF

[8] `http://dkm.fbk.eu/ontologies/knowledgestore#`

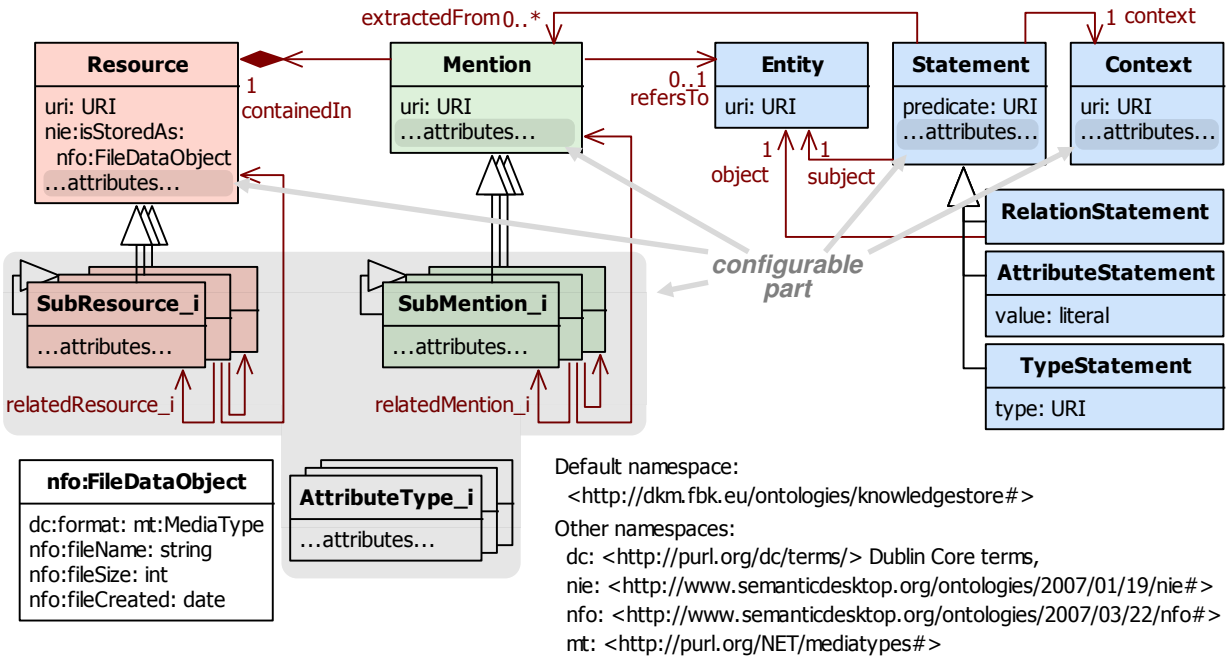

Figure 3: KnowledgeStore data model.

triples for statements and Named Graphs [Carroll *et al.*, 2005] for contexts.[9]

Fixed part This part defines the core abstractions and is embodied in the implementation, so it is kept as small as possible to increase the model flexibility. It includes:

- the Resource, Mention, Entity, Context classes and their uris, set at creation time and then immutable; entity uris are externally assigned, while the remaining are created by the system;
- the Statement class with the context relation and the subclasses, attributes and relations for encoding an entity type (TypeStatement - an entity is of a certain type), an entity attribute (AttributeStatement - a value associated to an entity) or a relationship among entities (RelationStatement - a relation between two entities);
- the files storing resource representations and their metadata managed by the system (nie:isStoredAs attribute and nfo:FileDataObject class);
- relations containedIn and refersTo linking a mention to the containing resource and the entity it possibly denotes;
- relation extractedFrom linking a statement to the mention(s) it has been extracted from; this information is relevant both for external users (e.g., decision makers) and for debugging an information extraction pipeline built on top of the KnowledgeStore.

[9]The system-generated context URI is used as the graph URI, while contextual metadata is encoded as additional triples describing the graph URI.

The formal specification of these elements is provided by a KnowledgeStore ontology, documented in Appendix B, that reuses terms from Dublin Core (dc:*)[10], the Nepomuk Information Element vocabulary (nie:*)[11] and the Nepomuk File Ontology (nfo:*)[12]. Note that relevant information such as a mention type, linguistic attributes or position in the containing resource are excluded, 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 the NewsReader one). Moreover, several aspects that can be fully controlled by the system and thus pertain to the fixed part, such as modification tracking and access control, are currently not considered; they will be added in the future, should the need arise.

Configurable part This part is specified at configuration time and is assumed to be available to the system, so that it can tune its storage options for improved efficiency and can possibly offer additional services (e.g., data validation).[13] It includes:

- the subclass hierarchy of Resource and Mention (entities excluded as described via statements), which are assumed not to be disjoint;
- additional attributes of Resource, Mention, Statement, Context and their subclasses (objects belonging to multiple subclasses are described using all their combined attributes); note that contextual attributes identify a context similarly to its URI;
- additional relations among resources or among mentions;
- enumerations and classes used as attribute types (similarly to nfo:FileDataObject);
- restrictions on the domain and range of fixed-part relations (not shown in figure).

The configuration exploits the alignment with OWL 2 and is performed by supplying the system with an OWL 2 ontology that imports the KnowledgeStore ontology and provides the TBox definitions for each configured element. Mention and resource subclasses, attributes, relations and auxiliary definitions and restrictions are read from this ontology. The choice of an OWL 2 ontology avoids to introduce a specific configuration format. Moreover, an ontology describing the data in the KnowledgeStore has to be produced anyway in case Linked Data publishing or sharing of instance data on the Semantic Web are desired.

2.2 Data model configuration for NewsReader

The UML class diagram in Figure 4 shows how the data model has been currently configured for NewsReader, based also on the inputs from WP3. The OWL 2 ontology formally

[10] `http://dublincore.org/documents/dcmi-terms/`

[11] `http://www.semanticdesktop.org/ontologies/nie/`

[12] `http://www.semanticdesktop.org/ontologies/nfo/`

[13] The alternative is represented by a completely free schema, unknown to the system and dynamically modifiable at runtime. Although more flexible, this solution has been discarded as it would require the use of a generic, non-optimized storage layout; moreover, a schema must already be available at design time in order for components to interact, so it makes sense for the KnowledgeStore to benefit from it.

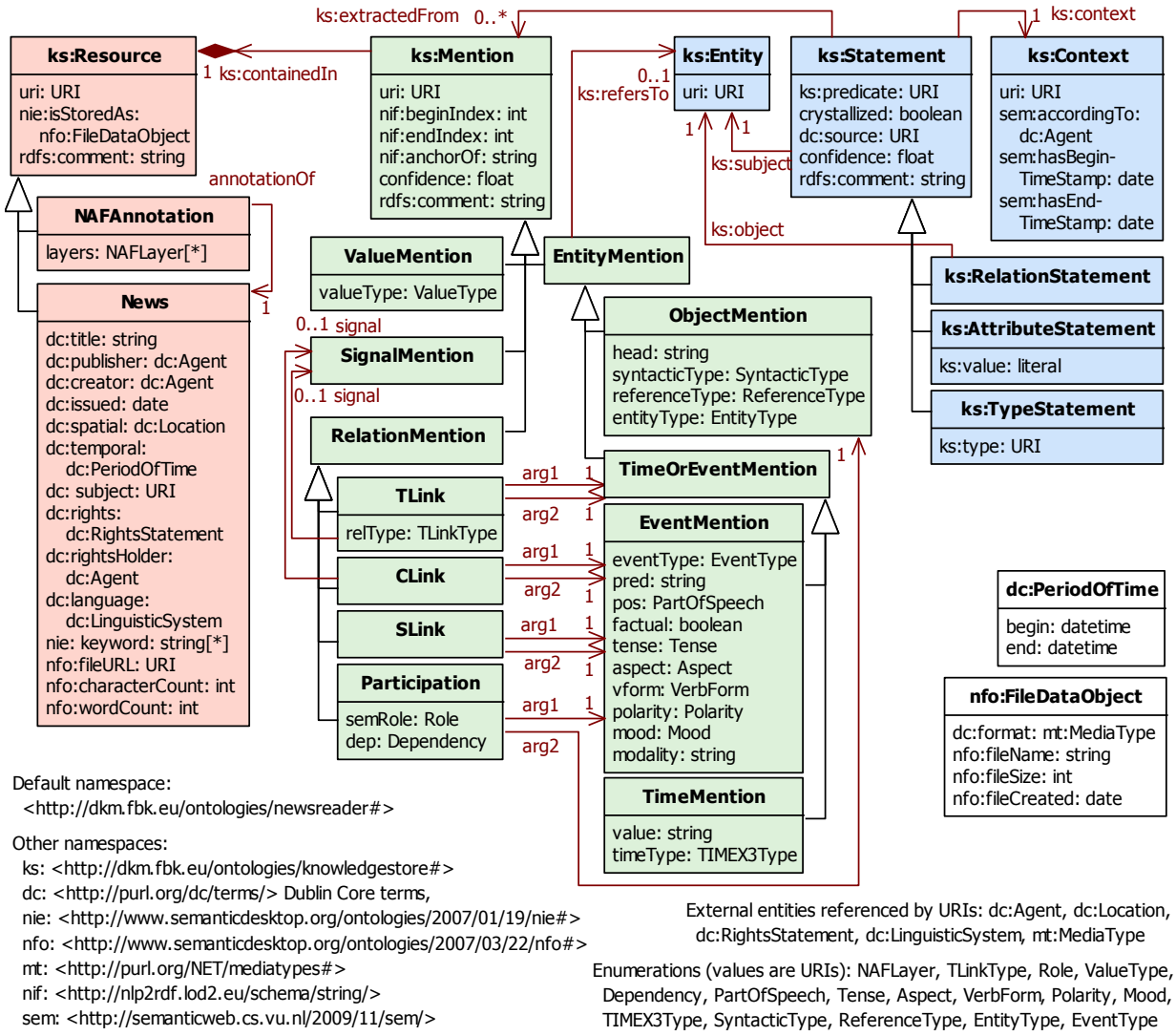

Figure 4: NewsReader data model.

encoding the model is reported in Appendix C. 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[14]).

Resource layer The resources of interest in NewsReader are News and NAFAnnotations[15]. News are described using metadata from the Dublin Core vocabulary (dc:* attributes), augmented with additional attributes to express keywords and the source file URL, when

[14] `http://dkm.fbk.eu/ontologies/newsreader#>`

[15] NAF (Newsreader Annotation Format) is the format adopted in the project to augment resources with structured information extracted by linguistic processors (tokenization, POS tagging, Semantic Role labelling, and much more). NAF is grounded in KAF (Kyoto Annotation Format) [Bosma *et al.*, 2009].

available; a character and a word count are also kept for statistical purposes. For NAF annotations only the list of annotated layers and the link to the annotated news are stored, as they are useful in accessing and selecting data in the KnowledgeStore, while annotation data is stored in the resource file.

Mention layer Mentions are represented according to the NAF specification[16]. The offset of a mention in a news, as well as its extent, are encoded using attributes from the NLP Interchange Format (NIF) vocabulary[17], thus enabling interoperability with tools consuming NIF data. A value (in a $0.0 - 1.0$ scale) can be assigned to a mention (attribute confidence) to represent the confidence of the linguistic processor on the extracted mention. Four main types of mentions are distinguished:

- *Entity mentions* denote entities in the domain of discourse (relation refersTo, restricted from fixed part), and are further characterized based on the type of entity. Object mentions refer to persons, locations, organizations, artefacts and financial objects (entityType), like e.g. "Barak Obama", "NASDAQ Index", "a family", "500 cars". The types considered are those proposed in Deliverable D3.1: Annotation Module. Object mentions are characterized by a mention head, a syntactic type (e.g., name, nominal or pronoun) and a reference type (e.g., specific referential). Time mentions are described by their TIMEX3 type and normalized time value. Event mentions are characterized using a number of NAF attributes: the lemma of the token conveying the event (pred); the part-of-speech (pos), e.g., adjective, noun or verb; the factuality of the event (factual); the tense, aspect, mood, verbal form (vform) of a verbal event; polarity (positive or negative) and modality (e.g. "should").

- *Relation mentions* express relations between two entities, whose mentions are identified by arg1 and arg2 or similar links. Different kinds of relation mentions are stored. Causal links (CLink) express a causal relation between two events, while temporal links (TLink) denote a certain temporal relation (relType, e.g., before, include, overlap) among two events or time expressions. Subordinate links (SLink) express certain structural relations among events. Participation mentions denote the participation of an entity to an event in a certain semantic role (semRole); the dep attribute denotes the syntactic dependency between the associated entity and event mentions.

- *Signal mentions* identify pieces of text supporting the existence of a causal or temporal 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.

[16]The NAF format and structure will be described in Deliverable D2.1: System Design - draft. The design of its content is documented in Deliverable D3.1: Annotation Module.

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

Entity layer Different kinds of entities are stored, including persons, organizations geo-political entities or locations, events, points and intervals in time extracted from text; the type of entity is conveyed by an rdf:type statement. The context in which a statement holds in is described and identified in terms of temporal validity (sem:hasBeginTimeStamp and sem:hasEndTimeStamp) and point of view (sem:accordingTo, e.g., "Financial Times"); the Simple Event Model (SEM) vocabulary [van Hage *et al.*, 2011] is used to that purpose. Statement metadata consists of a confidence value (confidence), a provenance indication (dc:source) and a crystallized flag (crystallized). Confidence is represented on a $0.0 - 1.0$ scale and quantifies how reliable is an extracted statement. Provenance is stored for a background knowledge statement and denotes the external source it has been imported from (e.g., DBPedia).[18] The crystallized flag is set for statements 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 it was extracted from (e.g., which kind of news) and how reliably; it will also consider pre-existing background knowledge, in form of TBox constraints a statement has to obey and other ABox statements in the background knowledge a statement has to be consistent with.

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

3 The KnowledgeStore Interfaces

The KnowledgeStore presents a number of interfaces through which external clients may access and manipulate stored data. While the physical realization(s) of these interfaces is sketched in Section 4.2.3 and depends on the overall NewsReader system design (WP2), the abstract definition of the API they implement and its rationale are described here. In particular, Section 3.1 introduces the criteria underlying the design of the API, touching the challenges that an API for Big Data access in a distributed setting must face. Section 3.2 presents an overview of the operations offered by the KnowledgeStore API: two main categories of operations are described, together with some representative examples. The full list of operations, instead, is reported in Appendix A.

3.1 API Design Criteria

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

Operation granularity An API may offer fine-grained, elementary operations operating on single objects (e.g., a single mention update), as well as coarse-grained operation that operate on whole sets of objects at a time (e.g., the simultaneous update of all the mentions of a certain resource). Some API designs allow for a coarse-grained operation even if redundant because equivalent to a sequence of fine-grained operations, on the basis of user convenience and/or improved performances, as a single API invocation and the associated overhead is then involved. For the KnowledgeStore API, the decision is instead to favour fine-grained operation wherever possible, in order to reduce the perceived API complexity as well as its implementation and maintenance cost. The overhead of long sequences of API calls is addressed by proper design at the protocol level (e.g., using techniques such as request pipelining) rather than introducing redundant coarse-grained operations.

Message exchange pattern One issue is the typology of operations provided by the KnowledgeStore with respect to the processing time and the amount of data returned. These two dimensions influence the message exchange pattern between server and client. The simplest case consists of operations characterized by low latency replies including little data; in this case the message pattern is the simple *request-response*: the client issues the request to the server and waits its reply. When the processing time sensibly increases, the client cannot block to wait the reply. In this case the adopted pattern is asynchronous and includes a sequence of steps: first, the client sends the request and immediately receives an acknowledge from the server that starts the processing. Then, there are two alternatives: (i) the client asks (possibly frequently) the server about the processing status and gets the data if finished (*asynchronous polling pattern*); or (ii) the server notifies the client when the processing is finished returning also the data (*asynchronous notification pattern*). With respect to the data size dimension, if large datasets have to be returned a streaming-based

protocol must be adopted, in order to avoid materializing full dataset in memory. Based on these considerations, suitable streaming solutions and polling / notification mechanisms will be selectively applied to API operations to deal, respectively, with the exchange of large quantities of data and with long running API operations.

Transactional properties Transactions are units of work—either a single operation or a sequence of operations—to which certain properties are associated, such as the *ACID* properties of relational databases: *atomicity*, *consistency*, *isolation* and *durability*.[19] Unfortunately, enforcing ACID properties in distributed, scalable systems like the KnowledgeStore is difficult, inefficient and even theoretically impossible in case system *availability* (i.e., the fact every request is answered) is also desired. With this premise, and assuming the need for *partition-tolerance* (due to the distributed nature of the system), the CAP theorem [Gilbert and Lynch, 2002] rules out consistency, and thus ACID in a strict sense.[20] The situation asks for a trade-off solution, that for the KnowledgeStore may favour consistency and ACID properties over availability, on the basis that it is deemed preferable for a client request to fail (in presence of nodes or network failures) rather than returning stale data. Defining the transactional properties of the KnowledgeStore, however, is difficult at the time of writing due to a lack of specific requirements and operational experience: a concrete solution will be developed in the course of the project, given feedbacks from client modules and the investigation of feasible approaches (such as the use of a distributed transaction manager). In the first KnowledgeStore release, single API calls will behave in a transactional way and satisfy ACID properties, as this can greatly simplify writing client applications. 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. 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 the future by exploiting the OWL 2 roots of the data model for declaring and validating complex constraints;[21] violations of these constraints may either be reported as warnings or may cause the API request to fail.

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

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

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

Security Access to the KnowledgeStore API must be restricted only to authorized clients, since it allows the modification of stored contents and the retrieval of possibly copyrighted or otherwise access-restricted information (e.g., news articles accessible only for research purposes). As it is conceivable for the KnowledgeStore API to be made accessible over an unprotected channel such as the Internet, suitable technical measures are implemented 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 Operations Categories

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

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

The collected operations were then first analysed[23] to find commonalities, in order to remove duplicates or operations subsumed by other ones. By adopting a generalization 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 Appendix A[24]. Here below, we present a brief overview of these operations, organized according to the content they are accessing: the content stored in a single layer (*intra-layer operations*), and the content stored across multiple layers (*inter-layer operations*).

can be fixed by automatically declaring objects in the KnowledgeStore as owl:differentFrom each other.

[22] `http://redmine.let.labs.vu.nl/projects/newsreader316404/wiki/KS_Access_Patterns`

[23] The content of the KnowledgeStore operations page on the project CMS may evolve during the lifetime of the project, as additional operations may arise with the development of new processing modules. The analysis here described refers to the content of the operations page available on 31.05.2013

[24] Also available at `http://redmine.let.labs.vu.nl/projects/newsreader316404/wiki/refactored_KS_Access_Patterns`

3.2.1 Intra-layer Operations

Operations dealing with content stored in a single layer of the KnowledgeStore are further organized in three main subcategories, as follows.

Operations on resources representations These operations allow manipulating the *representation* (i.e., the actual file) stored for a resource in the KnowledgeStore. Two operations are proposed: `storeResourceRepresentation()` stores the representation of a given resource defined in the system, while `retrieveResourceRepresentation()` enables its retrieval. As an example, we report the `retrieveResourceRepresentation()` operation:

`name`	`retrieveResourceRepresentation(resource URI) : representation`
`description`	retrieve a resource representation, if available
`input`	the `URI` of the resource (must exist in the KnowledgeStore)
`output`	the `representation` stored for the resource if any, otherwise an error signalling either that the URI is unknown or the representation is not available
`example`	retrieve the representation of news `nwr:r105`

CRUD operations on resources, mentions, entities, statements These operations refer to the possibility of creating (C), retrieving (R), updating (U) and deleting (D) elements stored in a single layer of the KnowledgeStore. They are provided for resources, mentions, entities and statements but not for contexts, which are referenced through their attributes in uploaded statements and are managed by the system (i.e., created when first referenced, deleted when no more used). CRUD operations are defined in terms of sets of objects, in order to enable bulk operations as well as operations on single objects. As an example, we report the operation for updating stored objects of type Mention:

`name`	`updateMentions(condition, mention, merge crit.) : update errors`
`description`	updates the mentions satisfying a condition, applying optional merge criteria
`input`	`condition` selecting the mentions to update; `mention description` with the attributes to set; `merge criteria` (optional) for (a subset of) the attribute URIs
`output`	for each non-updated mention (e.g., because of failed data validation), its URI and the error preventing the update
`example`	clear attribute `nwr:polarity` of all mentions of `type = nwr:EventMention`

SPARQL access to statements As the entity layer of the KnowledgeStore is grounded in Knowledge Representation and Semantic Web best practices, a further dedicated access mechanism will be provided to access in a flexible way (i.e., by means of queries in SPARQL[25], a standard query language able to retrieve and manipulate data stored in

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

Semantic Web repositories) entities and statements[26] stored in the KnowledgeStore.

Recall that, in our approach, each statement corresponds to a ⟨subject, predicate, object⟩ triple within a named graph [Carroll *et al.*, 2005] that encodes the context where the statement holds; the named graph is uniquely identified by the context URI associated to the statement (see Section 2.1). Therefore, clients interacting with the KnowledgeStore through SPARQL have to be aware of this contextual dimension when submitting the query to the KnowledgeStore, as well as when interpreting the results obtained.

Here below is the description of the `sparqlQuery()` operation offered by the KnowledgeStore:[27]

`name`	`sparqlQuery(query, dataset) : query solutions or triples`
`description`	evaluates the supplied SPARQL `query` on indexed statements or a subset of them identified by the `dataset` parameter
`input`	the `query` string, in the SELECT, ASK, CONSTRUCT or DESCRIBE forms; an optional `dataset` specification, consisting in a set of default graph URIs and named graph URIs (see FROM and FROM NAMED clauses)
`output`	on success, 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

`example`	evaluate the following query:
	`SELECT ?p ?e FROM nwr:ctx106 WHERE {`
	`  ?p a foaf:Person .`
	`  ?e a nwr:SellEvent ; sem:hasActor ?p }`

`notes`	results are streamed to the client

Before moving to inter-layer operations, let us note that intra-layer retrieval patterns dealing with entities/statements may require to exploit some automated reasoning (e.g., to retrieve all the events about sports requires to consider also subsumed event types such as football events).

3.2.2 Inter-layer Operations

These operations refer to the possibility to retrieve some objects by exploiting content distributed over two or more layers of the KnowledgeStore. Similarly to intra-layer operations, inter-layer retrieval operations dealing with entities/statements may require some automated reasoning (e.g., retrieve all the news where a sport event is mentioned). An example of inter-layer operations is the generalized `match()` operation, that exploits the dc:isPartOf and ks:refersTo relations between objects stored in different layers of the KnowledgeStore data model (see Figure 4) and can be used to navigate from resources to mentions and entities and back:

[26] To be more precise, as described in 4.2.2, only crystallized and background knowledge statements are accessible via SPARQL.

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

`name`	`match(condition and output attribute URIs at resource, mention and entity level) : matching <resource, mention, entity> tuples`
`description`	returns a set of ⟨resource, mention, entity⟩ tuples whose mention occurs in the resource and refers to the entity, and such that their attributes and statements satisfy a certain condition; for each tuple, a specified set of output attributes for each resource, mention and entity element is returned
`input`	a condition over the attributes of matched resources and mentions and over the statements of matched entities; a list of predicate URIs identifying output attributes for resources and mentions, and output statements for entities
`output`	an unordered list of matching ⟨resource, mention, entity⟩ tuples with the output attributes specified
`example`	retrieve `uri` and `dc:title` of all the resources of type `nwr:News` in which entity `dbpedia:Berlin` is mentioned
`notes`	results are streamed to the client

It is worth mentioning that while most of the inter-layer operations are tailored to the specific requirements of NewsReader modules, intra-layer and the `match()` operations are general, and easily hold also in application scenarios different from NewsReader. When considering different application scenarios, an interesting family of operations consists in the full text search of resources, mentions and entities, whose content (for resources) or attributes (for all kinds of objects) are matched to keyword queries; an example of such a query is "return all the news whose content (or title) matches "crisis OR recession". While potentially relevant in the general case, full text search has not emerged as a requirement for NewsReader so far. If necessary, it may be implemented either internally to the KnowledgeStore, or externally by some dedicated component in the broader scope of the NewsReader architecture (either inside or outside WP6).[28] Given the goal to develop a general purpose KnowledgeStore component—reusable outside the NewsReader scenario—we plan to further investigate its extension with full text search during the project lifetime.

We conclude this section with a remark on the expected performances (in terms of execution time) of the methods implementing the KnowledgeStore operations. While intra-layer operations can easily scale (e.g., CRUDs), other operations, and in particular the inter-layer ones, may be potentially affected by performance issues due to their greater complexity and the (possible) involvement of automated reasoning. To mitigate these issues, especially for operations requiring a "real-time" response, appropriate techniques need to be implemented (e.g., denormalization and additional indexes to avoid expensive joins), as reported in Section 4.2.1.

[28]The task of such a component would consist in realizing a full text index of (part of) the contents stored in the KnowledgeStore; to that purpose, scalable open source products such as Elasticsearch (`http://www.elasticsearch.org/`) or Apache Solr (`http://lucene.apache.org/solr/`) can be adopted. In case an internal implementation is preferred, these products can be embedded in the KnowledgeStore and their functionalities accessed by extending the `conditions` used in the KnowledgeStore API for matching resources, mentions and entities with a "full text match" primitive.

4 The KnowledgeStore Architecture

This section presents the KnowledgeStore architecture. From a functional point of view, the KnowledgeStore is a server whose role within the NewsReader system and whose typical clients are introduced in Section 4.1. Section 4.2 describes the KnowledgeStore internal architecture, discussing the design and implementation of the main components for the storage & retrieval of resources, mentions and entities.

4.1 Architectural overview

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.

It is worth noticing here that the KnowledgeStore is a passive component, without any active role concerning the orchestration of the other NewsReader modules. The interaction between the KnowledgeStore and an external orchestrator may occur in two forms (in a way similar to the "Message exchange pattern" discussed in Section 3.1):

1. *by polling*: an external orchestrator (or even each processing module) periodically asks the KnowledgeStore whether there is something new to process (e.g., based on data-stamps or other metadata attached to the content of the KnowledgeStore);
2. *by notification*: the external orchestrator is notified by the KnowledgeStore after every modification and may trigger processing modules; different notification mechanisms can be adopted. For example, within an orchestration infrastructure such as the Storm framework[29], the KnowledgeStore may interface with it using a notification approach based on a message queue where to emit modification events (e.g., "resource added", "mention modified", ...)

The appropriate external orchestration interaction choice will be defined (if needed) in light of the general NewsReader system architecture (defined in WP2).

From a functional point of view, clients can be classified in two main categories, according to the type of operations used to interact with the KnowledgeStore: *populators* and *applications*. Populators are clients whose main purpose is to feed the KnowledgeStore with new data. They play an important role into the NewsReader system, since they write into the KnowledgeStore the basic contents needed by applications, such as the Resources supplied by data providers and the background knowledge for the Entities[30]. Instead, applications are clients that (mainly) read data from the KnowledgeStore to accomplish their tasks. An example of such applications is the decision support system. Some applications also store the results of their processing back into the KnowledgeStore (e.g., linguistic processors, tools supporting knowledge crystallization).

[29] `http://storm-project.net/`

[30] More specialized populators can be realized to store more peculiar contents such as custom annotations provided by specific linguistic processors: in this case linguistic processors providing this specific content are in charge of proving custom populators.

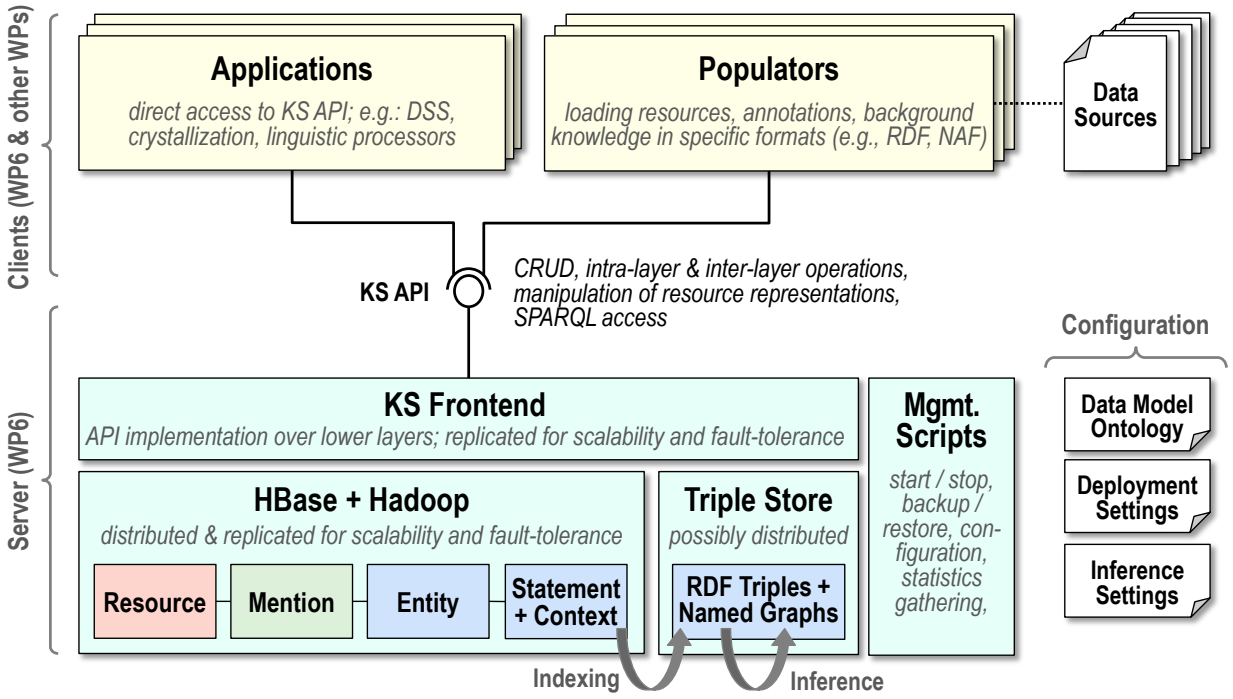

Figure 5: KnowledgeStore architecture.

4.2 KnowledgeStore internal architecture

The lower part of figure 5 shows the KnowledgeStore internal architecture. There are three main components, which are detailed in the remainder of the section:

- *the HBase & Hadoop component*: this module stores all the information at the resource (including physical files), mention and entity levels, providing typical database services; semantic queries are not supported by this component;
- *the Triple Store*: crystallized and background knowledge statements are indexed by this module to provide services supporting reasoning and online semantic query answering, which cannot be easily and efficiently implemented in HBase or Hadoop;
- *the KS Frontend*: this component provides the API implementation of the KnowledgeStore operations; it exposes the services realized by the two previous modules, including mixed queries, and it is in charge of global issues (e.g., transactions and data validation, when implemented).

It is worth noticing here that, for the sake of scalability, the KnowledgeStore is expected to be deployed on a cluster (potentially in a cloud environment), therefore additional tools will be implemented to deal with the complexity of such deployment. This includes, for example, management scripts for operations at system level such as infrastructure (daemons) start-up & shut-down, data backup & restoration, statistics gathering.

4.2.1 The HBase & Hadoop component

Hadoop[31] and HBase[32] are frameworks developed by Apache to manage scalability for file systems and databases, respectively. Distributed computation on multiple nodes, replication and fault tolerance with respect to single node failure are their key features. HBase is particular suited for random, real time read/write access to huge quantity of data (such as *big data*), when the data's nature does not require a relational model. HBase belongs to the *NoSQL* database family: it provides a mechanism for storage and retrieval of data that use looser consistency models than traditional relational databases in order to achieve horizontal scaling and higher availability. It does not (natively) support SQL-like queries.

The KnowledgeStore utilizes the Hadoop distributed file system (DFS) to store resource representations, that is the physical files such as news documents or custom annotations provided by the linguistic processors. HBase is used as a database to store the remaining information, with dedicated tables used to store resource metadata, mentions, contexts and statements attributes. The last type of object—statement—deserves a special description, as they are also stored in the Triple Store component, but with different purposes. Full information about statements is stored in the HBase & Hadoop component, where each statement is stored as a tuple holding its ⟨subject, predicate, object, context⟩ components and statement-level metadata attributes (e.g., provenance and confidence values), as shown in the top part of Figure 6. This solution allows for a compact representation of statement metadata and fast lookup by subject among huge collections of statements, but it's not enough for supporting SPARQL queries [Seaborne and Harris, 2013], which are instead provided by the Triple Store component. The latter holds only a subset of the statements (e.g., only the crystallized ones, so to reduce the load on the Triple Store), and just stores their ⟨subject, predicate, object, context⟩ components, with the first three encoded as an RDF triple and the latter as the named graph containing the triple, as shown in the bottom part of Figure 6. Statement metadata is not stored, as (i) it is often irrelevant to user queries, and (ii) its representation would require the use of expensive and impractical techniques such as RDF reification.[33]

The design of HBase tables is crucial for achieving good performances. In particular two aspects are to be taken into account: definition of the row keys, which deeply impacts on the random access of a table, and denormalization. Concerning row key definition, some examples of smart row key design that we will adopt in the KnowledgeStore are:

- defining the resource key as dc:date | rdf:type | progressive number[34], in order to allow fast retrieval of resources published in a certain period and possibly of a specific type;
- defining mention key as key(ks:resource) | nif:beginIndex, in order to allow fast retrieval of all mentions of a given resource (as its key is a prefix of the mention row key);

[31] `http://hadoop.apache.org`

[32] `http://hbase.apache.org`

[33] Note that in the KnowledgeStore implementation it is always possible to go back and forth from one representation to the other, since statements are uniquely identified by their ⟨subject, predicate, object, context⟩ components which are stored both in HBase & Hadoop and in the Triple Store.

[34] Symbol | denotes a concatenation operation

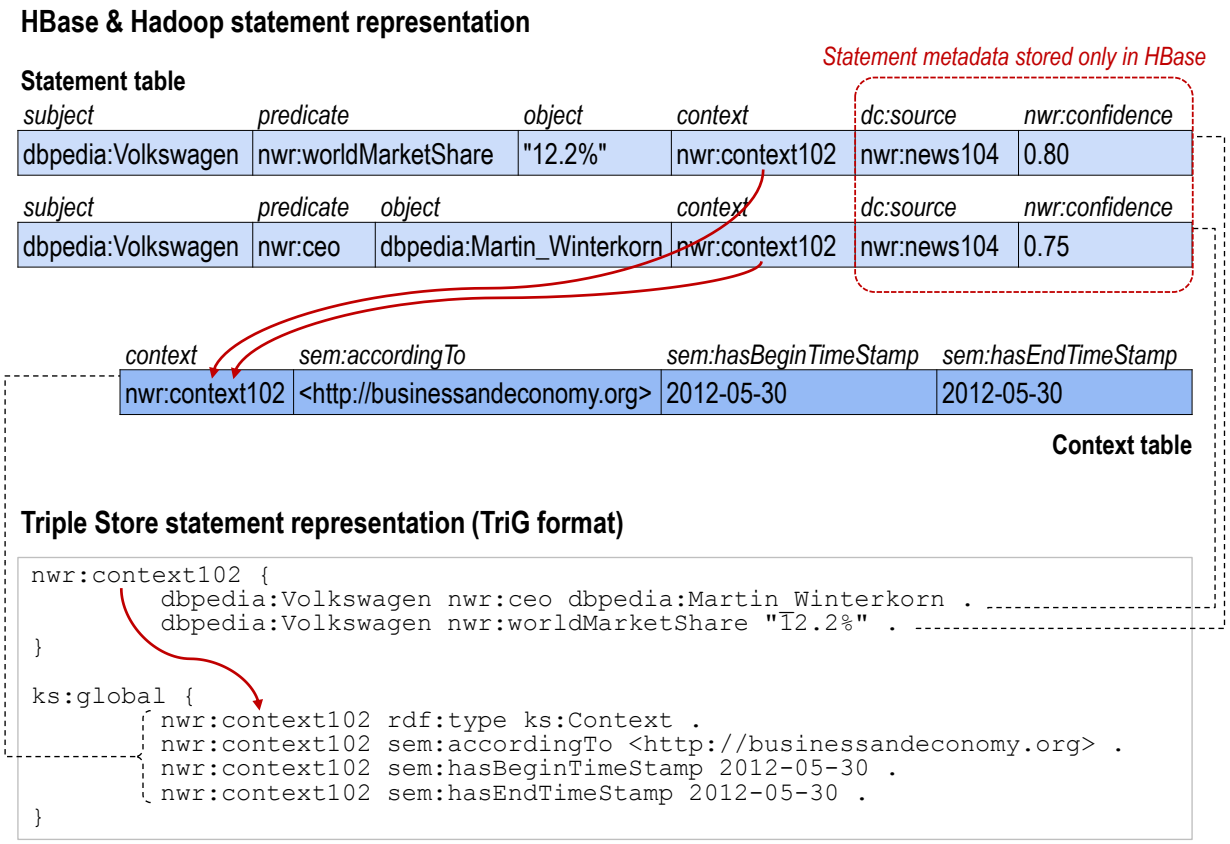

Figure 6: Statement representation in HBase & Hadoop and Triple Store components.

- defining statement key as key(ks:subject) | hash(ks:subject, ks:predicate, ks:object, ks:context), in order to allow fast retrieval of all the statements for an entity.

Concerning denormalization, that is the replication of partial information on redundant attributes or tables, in the KnowledgeStore we are considering the following proposals:

- to achieve fast retrieval of the resources mentioning an entity, a redundant "entity-to-resource" table can be added, whose row key is key(entity) | key(mention) (the latter including the resource row key);
- to achieve a controlled denormalization a possible approach is to distinguish between *summary* and *detail* attributes of an element, with the assumption that the first attributes are more frequently accessed than the latter. Summary attributes of an element are then replicated in the rows of all other elements referring to it, in order to avoid a join operation each time the referring object is retrieved.

It is worth noting that denormalization impacts on the transactionality of KnowledgeStore API operations, as it requires multiple HBase calls to update the master copy of data as well as its replica. While for a single call HBase guarantees transactionality, in case of multiple HBase calls it must be manually guaranteed by the KnowledgeStore.

Another issue to be taken into account is the implementation of intra- and inter-relations between layers in HBase. An example of the former is the nwr:annotationOf relation linking an annotation resource to the news resource it derives from (see Figure 4); an example of the latter is the relation ks:refersTo connecting a mention to the entity it denotes. From an implementation point of view, we are considering several approaches with different read, storage and implementation costs, but all able to store N:N relations with attributes:

- *Blob approach.* If no indexing of relations is needed, a "blob" field can be added to one (or both) of the involved HBase rows (e.g., the one for the resource or mention) where to store all the associated instances. Benefits of this solution include space efficiency and transactional update of relationships (if accessing a single row).
- *Wide table approach.* Several HBase columns can be added to one or both of the involved object, such as $\mathsf{mention}_1$, ... , $\mathsf{mention}_N$. Benefits of this solution include a transactional modification of related objects and faster updates of related objects (no need to rewrite unchanged related objects as in the blob approach).
- *Long table approach.* Given a relation between object_1 and object_2 (being, e.g., resources or mentions), an HBase table is created whose row key is $\mathsf{key}(\mathsf{object}_1)$ | $\mathsf{key}(\mathsf{object}_2)$, and whose columns are the attributes of the relation, plus possibly redundant attributes copied here to speed up read access. Benefits of this solution include efficient storing of attributes, fast retrieval of instances associated to a given instance, fast checking of whether a relationship holds, fast updates and reduction of table number (a single HBase table can be used for multiple relations and even their inverses if the relation type is included in the row key).

The most suitable option, as well as the actual definition of the HBase tables (number, structure and row key definition), will be chosen according to the operations presented in Section 3.2 (and their further refinements as required by the KnowledgeStore clients), and in particular those used by applications that need almost real time access.

The aspects discussed so far highlight that, especially to guarantee acceptable performances, the HBase component of the KnowledgeStore has to be optimized with respect to the operations required by the NewsReader modules: a change in such operations may require to re-think and adapt its design and, therefore, implementation.

4.2.2 The Triple Store

Statements are indexed in a triple store so to enable efficient, inference-aware SPARQL-based query answering. Indexing affects only those statements that satisfy certain configurable criteria; this allows, for instance, to exclude from inference non-crystallized statements or statements whose extraction confidence level is below a given threshold.

As remarked in Section 3.2.1 and Section 4.2.1, each statement is stored as a ⟨subject, predicate, object⟩ triple within a named graph [Carroll *et al.*, 2005] that encodes the context where the statement holds. Contexts are defined by additional triples (placed in a *global* graph) that define their attributes such as the point of view (sem:accordingTo) and time validity (sem:hasBeginTimeStamp and sem:hasEndTimeStamp) identifying the context.

$$\frac{\texttt{?ctx \{ ?x rdf:type ?c1 \}} \quad \texttt{ks:global \{ ?c1 rdfs:subClassOf ?c2 \}}}{\texttt{?ctx \{ ?x rdf:type ?c2 \}}}$$

(a) Contextual version of RDFS9

$$\frac{\texttt{?ctx1 \{ ?s ?p ?o \}} \quad \texttt{ks:global \{ ?ctx2 skos:broader ?ctx1 \}}}{\texttt{?ctx2 \{ ?s ?p ?o \}}}$$

(b) Propagation from broader contexts

Figure 7: Examples of inference rules

As anticipated, additional statement metadata are not indexed. By resorting to a triple store, indexed statements can be easily queried and manipulated using SPARQL both as a language and access protocol (via so-called SPARQL endpoint offered by triple stores).

Logical inference aims at deriving the additional statements implied by stored data (ABox) and the ontologies defining its schema (TBox), and making them available as possible answers to applications and users queries. For instance, if a statement describes dbpedia:Volkswagen as a nwr:PublicCompany and nwr:PublicCompany is a subclass of nwr:Company in the KnowledgeStore background knowledge, then a query for all companies (e.g., from the decision support suite) is expected to return dbpedia:Volkswagen as an answer. Inference in the KnowledgeStore must take into consideration the large amount of data available as well as its contextual validity: the first aspect asks for a scalable and efficient approach, such as the off-line pre-materialization of the logical closure that speeds up online query answering; the second aspect asks for a custom solution, such as a custom set of inference rules, as no standardized ontological language currently supports reasoning with contextualized data. Figure 7 shows two examples of such inference rules. Figure 7a shows the contextual extension of rule RDFS9 (the rule responsible for the dbpedia:Volkswagen inference example reported above), which is applied on a per-context basis using TBox definitions (the rdfs:subClassOf triples) declared in a global context (ks:global); Figure 7b shows how statements in a context can be propagated to other contexts if the former is declared (or found, via inference) to be broader in scope, thus implementing a semantics according to which statements in a context hold also in narrower contexts. Although logical inference is a task for the second year of the project (T6.3, starting month 15), it is worth noticing here that techniques such as closure materialization and rule-based reasoning can be efficiently implemented in a triple store[35], thus further justifying its inclusion in the KnowledgeStore architecture.

Different triple store implementations offer different performance, scalability and fault-tolerance characteristics, as well as licenses (e.g., open source vs commercial). The initial implementation of the KnowledgeStore will be based on the Open Source Edition of the Virtuoso triple store[36], a product showing excellent performances in recent (April 2013) benchmarks[37]. The Open Source Edition is limited to a single node deploy, where it can

[35]Rule-based reasoning is directly supported by some triple stores; if not supported, it can be implemented through the fix-point evaluation of SPARQL queries.

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

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

easily handle a billion of triples.[38] Additional scalability and transparent fault tolerance can be obtained using the (commercial) Enterprise Edition, or moving to alternative open source clustered triple stores. The use of the OpenRDF Sesame Java API[39] will permit the KnowledgeStore to largely abstract from the adopted triple store implementation[40], thus allowing to change it within (and beyond) the scope of NewsReader.

4.2.3 The Frontend

The Frontend component implements the external API of the KnowledgeStore by dispatching client requests to the appropriate internal components, and by implementing the necessary streaming, notification and/or polling mechanism discussed in Section 3.1. It also controls the indexing of statements in the Triple Store component and the triggering of inference, which are transparently performed each time data is written through the API.

The majority of API operations is forwarded to a single component, either HBase & Hadoop or the Triple Store; they include all the intra-layer operations as well as several inter-layer operations. The remaining inter-layer operations are *mixed queries* which can be decomposed into one or more semantic queries, targeted at the Triple Store, and one or more retrieval operation for structured and unstructured data stored in Hadoop & Hbase. For those operations, it is a responsibility of the Frontend to orchestrate their execution, by invoking internal components in the proper order, filtering and composing the final result.

Efficient communication protocols and compact data formats are crucial for the API implementation in the Frontend. Their selection will be performed in the scope of WP2 consistently with the protocols and formats used for communication among other NewsReader components. Possible choices include (but are not limited to) an HTTP Rest API, a WSDL+SOAP web service API as well as more efficient binary Remote Procedure Call (RPC) mechanisms such as Apache Thrift[41]. While a Rest API is more faithful to the Web architecture, the remaining approaches may be more efficient and allow for the automatic and cross-language generation of the client and server implementations of the API, based on an abstract specification written in an Interface Description Language (IDL). The KnowledgeStore Frontend will also expose the SPARQL endpoint of the Triple Store component, to provide external clients and (Semantic Web) applications a flexible and popular access mechanism to indexed statements.

In order to avoid single points of failure, the Frontend component will be replicated, taking inspiration from the approach utilized by the HBase and Hadoop daemons. A possible solution to achieve this consists in storing all the system state in HBase and deploying a suitable locking mechanism (or even a transaction manager, such as OMID[42]) to coordinate the access of multiple Frontend instances to such state.

[38] As a reference, the English DBPedia consists of 400M triples about 3.77M entities, and can be loaded as background knowledge in the triple store leaving room for a similarly sized volume of crystallized data.

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

[40] Methods for efficient bulk data ingestion are specific to each triple store implementation.

[41] `http://thrift.apache.org/`

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

5 Related Work

In this section, we review some state-of-the art contributions related to the KnowledgeStore approach for storing and managing, in an integrated and interlinked way, unstructured and structured content. We also discuss some state-of-the-art technologies, relevant for the implementation of the KnowledgeStore architecture.

5.1 Related approaches

The development of frameworks able to store integrated and interlinked unstructured and structured content was not deeply explored in the literature.

Some investigations were carried out on document repositories based on semantics (e.g., [Bang and Eriksson, 2006; Eriksson, 2007]). In these approaches, ontologies are used to represent domain vocabulary and the document structure, and they are used to annotate documents and document parts. However, the repository adopting these approaches (i) emphasise the document structure (e.g. tables, title) rather than document content, (ii) they do not foresee integrated framework for storing semantic content and unstructured documents altogether, and (iii) they are typically not meant to be applied in big data contexts.

Relevant for our work is the contribution presented in [Croset *et al.*, 2010]. The authors present a framework, based on a RDF triple store, 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 multimedia content (triple stores currently provide only a limited support for integrating knowledge with unstructured resource, often consisting in simple full text search capabilities on RDF literals), and the framework is focused only on name entities appearing in the unstructured content (i.e., occurrences of events and relation between entities of the domain were not covered).

Of some relevance are also frameworks for the extraction and storage of knowledge from Wikipedia, although they focus on providing a knowledge base and not an interlinked structured / unstructured knowledge source. For instance, in [Hoffart *et al.*, 2013] the authors of YAGO 2 describe a framework to represent contextualized knowledge extracted from Wikipedia. Facts, mainly contained in Wikipedia Infoboxes, are enriched with three additional dimensions (time, location, context), and the whole content (447 million facts, 9.8 million entities) is stored in a relational database (PostgreSQL). However, this framework does not support the interlinking of an entity with (the exact position within) the document where it is mentioned (the mention layer in our approach), as only the extracted information is stored.

Although exploited in a different context, dealing with much smaller quantity of content, also semantic desktop applications (e.g., MOSE [Xiao and Cruz, 2006], Nepomuk [Groza *et al.*, 2007][43]) are partly related with contribution here presented. Semantic desktop

[43]Nepomuk EU project, `http://nepomuk.semanticdesktop.org/`

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.

A repository envisioned in [Petrov, 2013] to support large-scale language learning shares (at a very abstract level) a similar three layers internal organization as the one detailed in the KnowledgeStore data model.

5.2 Related technologies

The KnowledgeStore combines Hadoop, HBase and the Virtuoso triple store to provide a platform for the storage of large volumes of interconnected unstructured and structured data, augmented with inference and semantic query facilities. In the following, some technological solutions for supporting these tasks are briefly illustrated, motivating the adoption of certain solutions for the KnowledgeStore implementation.

Storage of structured and unstructured contents If approached separately, storing structured and unstructured contents can be tackled with several solutions – both open source and commercial. For the former, *NoSQL* databases[44] are gaining popularity in contexts where huge data are to be managed with commodity hardware by means of horizontal scalability and fault-tolerance. In the case of unstructured data[45], the available solutions focus on systems and/or architecture for analysing and understanding unstructured data for business applications. In this respect UIMA[46] is a framework developed by IBM and implemented also by Apache[47]. In the case of the KnowledgeStore, the choice of Hadoop & HBase is rather natural. First of all they were specifically designed to manage big data by means of fault tolerance and horizontal scalability. Then, together they present an integrated solution to save both unstructured and structured data: the former with Hadoop HDFS, the latter with HBase tables. Moreover, the Hadoop and HBase infrastructure natively supports the *map-reduce* computational paradigm, a feature that could be exploited by the NLP processors and other applications (e.g., Decision Support System).

Inference and semantic queries Semantic queries in the KnowledgeStore are expressed in SPARQL and are evaluated on the RDF data of the entity layer. As any other type of query, semantic queries require particular indexing of data in order to be evaluated efficiently; in addition, they may require preprocessing of data in order to take inference into consideration. The need of data indexing and preprocessing and to deal with the complexity of SPARQL rules out a custom implementation on top of the data structures already in place in the HBase part of the KnowledgeStore (implementing SPARQL over HBase or

[44] http://en.wikipedia.org/wiki/NoSQL
[45] http://en.wikipedia.org/wiki/Unstructured_data
[46] http://en.wikipedia.org/wiki/UIMA
[47] http://uima.apache.org/

other column stores has already been attempted [Khadilkar *et al.*, 2012], but inference support and mature products are still lacking and their implementation in NewsReader is out-of-scope). Rather, the requirements suggest the integration in the KnowledgeStore of a *triple store*, as it represents the state of the art in managing and querying RDF data.

Different triple store implementations are available, either commercial or open source and with different support for inference (see next). They can be roughly classified in native, RDBMS and NoSQL based implementations, based on whether RDF data is stored in custom indexes (as in Owlim[48], Bigdata[49], 4Store[50]), a relational database (as in Virtuoso and the *Oracle Spatial and Graph RDF Semantic Graph*[51], usually using a ⟨*subject*, *predicate*, *object*, *context*⟩ table) or a NoSQL database (e.g., a graph database such as Neo4J[52]). While Virtuoso will be used in the current implementation due to good performances and open source availability, several triple stores are compatible with the KnowledgeStore, and hence the use of a standard-access API such as Sesame or Jena[53] is mandatory.

The distinguishing aspect of semantic queries is their reliance on inference for returning a complete set of answers that consider implicit information not directly encoded in asserted triples. Two approaches to inference are commonly adopted: *forward-chaining* and *backward-chaining*. Forward-chaining inference is adopted by the majority of triple stores (e.g., Owlim, Bigdata, Oracle) and is performed offline, augmenting stored data with all the consequences that can be inferred from it; semantic queries are then evaluated by simply matching their body with augmented data, without further considering inference. Backward-chaining inference is performed at query time, by matching each clause of a query in a way that consider not only asserted data, but also data that can be inferred from it. This is performed either by rewriting the original query (e.g., a query for nwr:Managers will be rewritten as a query for nwr:Managers, nwr:CEOs, nwr:CTOs, ... if the latter are subclasses of the first), or by recursively evaluating its clauses with other queries that takes all the possible inference paths into consideration. Backward-chaining is employed in some triple stores (e.g., Virtuoso) as well as in *Ontology-Based Data Access* systems [Calvanese *et al.*, 2007], but the latter employ a fixed, relational schema to store data, backed by a RDBMS, and cannot thus be used to store arbitrary RDF data as needed in the KnowledgeStore. Forward-chaining appears more convenient for the KnowledgeStore, as it provides for better query performances. On the down side, it adds overhead in terms both of space for materialized data and time for data modification operations, which need to update materialized inferences as well; in particular, forward-chaining copes badly with data deletion—an infrequent operation in the KnowledgeStore—which implies a retraction of selected inferences (if determinable) or even their complete recomputation. Nevertheless, in the scope of the LarkC research project[54] forward-chaining has been shown to scale

[48] `http://www.ontotext.com/owlim`
[49] `http://www.bigdata.com/bigdata/blog/`
[50] `http://4store.org/`
[51] `http://www.oracle.com/technetwork/database-options/spatialandgraph/`
[52] `http://www.neo4j.org/`
[53] `http://jena.apache.org/`
[54] LarkC (`http://www.larkc.eu/`) developed a workflow-based approach where different reasoner plu-

to huge datasets using *MapReduce* techniques [Urbani *et al.*, 2012].

Independently from a forward vs backward approach, it is worth noting that out-of-the-box inference in triple stores is usually restricted to standard ontological languages with a rule-based implementation, such as RDFS, OWL-Horst [ter Horst, 2005] and OWL 2 RL[55], whereas an approach considering also named graphs (i.e., contexts) is required for the KnowledgeStore (see Section 4.2.2). This limitation is shared also by external DL reasoners that can be layered over a triple store, and by the scalable reasoners developed in the LarkC project, such as WebPie [Urbani *et al.*, 2012] and QueryPie [Urbani *et al.*, 2011], which are restricted to OWL, and the distributed IRIS reasoner[56], which is restricted to RDF without named graphs. A customization of the inference engine within the triple store (e.g., by reconfiguring its inference rules), or an adaptation of one of the scalable reasoners cited above is thus required.

gins can be integrated to process large amounts of RDF data in a distributed fashion; both the workflow engine and several specialized reasoners (either integrable in the workflow or separate) were released.

[55] `http://www.w3.org/TR/owl2-profiles/#OWL_2_RL`

[56] `https://github.com/distributed-iris-reasoner/distributed-iris-reasoner`

6 Conclusions and Future Work

In this deliverable we documented the design of the 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., resource/mention/entity processors, decision support system) cooperate. The description of the KnowledgeStore design will also appear in the proceedings of the 7th IEEE International Conference on Semantic Computing (ICSC2013) [Rospocher *et al.*, 2013].

The implementation of the KnowledgeStore according to the presented design criteria is currently in progress. We are following an iterative development process, so that a first release of the KnowledgeStore, implementing the core functionalities of the framework, will be progressively refined and extended with additional features.

During the course of the project, we plan to evaluate the KnowledgeStore from a functional perspective based on its capability to (i) store an overwhelming daily stream of economical and financial contents (news articles and data), (ii) support a complex NLP pipeline in extracting knowledge from those contents, and (iii) provide suitable online and offline query capabilities for use in a decision support tool for professional decision-makers. In the same context, we also plan to carry out an extensive performance evaluation in terms of scalability with respect to data size, query load, and tolerance to nodes and network failures.

References

[Bang and Eriksson, 2006] Magnus Bang and Henrik Eriksson. Towards document repositories based on semantic documents. In *Proc. of 6th Int. Conf. on Knowledge Management and Knowledge Technologies (I-KNOW'06)*. Springer, 2006. http://i-know.tugraz.at/papers/towards-document-repositories-based-on-semantic-documents.

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

[Bosma *et al.*, 2009] Wauter E. Bosma, Piek Vossen, Aitor Soroa, German Rigau, Maurizio Tesconi, Andrea Marchetti, Monica Monachini, and Carlo Aliprandi. KAF: a generic semantic annotation format. In *Proc. of the GL2009 Workshop on Semantic Annotation, Pisa, Italy*, September 2009. http://www.researchgate.net/publication/228922488_KAF_a_generic_semantic_annotation_format/file/9fcfd50a355ff71027.pdf.

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

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

[Calvanese *et al.*, 2007] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. Ontology-based database access. In *Proc. of the 15th Italian Symposium on Advanced Database Systems (SEBD'07)*, pages 324–331, June 2007. http://www.dis.uniroma1.it/~poggi/publi/SEBD2007.pdf.

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

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

[Croset *et al.*, 2010] Samuel Croset, Christoph Grabmüller, Chen Li, Silvestras Kavaliauskas, and Dietrich Rebholz-Schuhmann. The CALBC RDF triple store: Retrieval over large literature content. In *Proc. of the Workshop on Semantic Web Applications and Tools for Life Sciences (SWAT4LS), Berlin, Germany*, volume 698 of *CEUR Workshop Proceedings*. CEUR-WS.org, December 2010. http://ceur-ws.org/Vol-698/paper6.pdf.

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

[Eriksson, 2007] Henrik Eriksson. The semantic-document approach to combining documents and ontologies. *Int. J. Hum.-Comput. Stud.*, 65(7):624–639, July 2007. http://dx.doi.org/10.1016/j.ijhcs.2007.03.008.

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

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

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

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

[Glimm and Ogbuji, 2013] Birte Glimm and Chimezie Ogbuji. SPARQL 1.1 entailment regimes. Recommendation, W3C, March 2013. http://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/.

[Groza *et al.*, 2007] Tudor Groza, Siegfried Handschuh, Knud Moeller, Gunnar Grimnes, Leo Sauermann, Enrico Minack, Cedric Mesnage, Mehdi Jazayeri, Gerald Reif, and Rosa Gudjonsdottir. The nepomuk project - on the way to the social semantic desktop. In *Proc. of 3rd Int. Conf. on Semantic Technologies (I-SEMANTICS'07), Graz, Austria*, pages 201–211. JUCS, September 2007. http://www.dfki.uni-kl.de/~sauermann/papers/groza+2007a.pdf.

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

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

[Khadilkar *et al.*, 2012] Vaibhav Khadilkar, Murat Kantarcioglu, Bhavani M. Thuraisingham, and Paolo Castagna. Jena-HBase: A distributed, scalable and efficient RDF triple store. In *Proc. of Int. Semantic Web Conference – Posters & Demonstrations Track (ISWC'12), Boston, USA*, volume 914 of *CEUR Workshop Proceedings*. CEUR-WS.org, November 2012. http://ceur-ws.org/Vol-914/paper_14.pdf.

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

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

[Petrov, 2013] Slav Petrov. Large-Scale Language Learning. Technical report, Google, Slides presented at the "Big data: theoretical and practical challenges" workshop, May 15, 2013. http://bigdata2013.sciencesconf.org/conference/bigdata2013/pages/petrov.pdf.

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

[Seaborne and Harris, 2013] Andy Seaborne and Steve Harris. SPARQL 1.1 query language. Recommendation, W3C, March 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

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

[ter Horst, 2005] Herman J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *J. Web Sem.*, 3(2-3):79–115, October 2005. http://dx.doi.org/10.1016/j.websem.2005.06.001.

[Urbani *et al.*, 2011] Jacopo Urbani, Frank van Harmelen, Stefan Schlobach, and Henri Bal. QueryPIE: backward reasoning for OWL Horst over very large knowledge bases. In *Proc. of the 10th Int. Semantic Web Conference (ISWC'11), Bonn, Germany*, pages 730–745, Berlin, Heidelberg, 2011. Springer-Verlag. http://dl.acm.org/citation.cfm?id=2063016.2063063.

[Urbani *et al.*, 2012] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank van Harmelen, and Henri E. Bal. WebPIE: A web-scale parallel inference engine using MapReduce. *J. Web Sem.*, 10:59–75, 2012. http://dx.doi.org/10.1016/j.websem.2011.05.004.

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

[Xiao and Cruz, 2006] Huiyong Xiao and Isabel F. Cruz. Application design and interoperability for managing personal information in the semantic desktop. In *Proc. of the Semantic Desktop and Social Semantic Collaboration Workshop (SemDesk'06) located at the 5th Int. Semantic Web Conference (ISWC'06), Athens, GA, USA*, volume 202 of *CEUR Workshop Proceedings*. CEUR-WS.org, November 2006. http://ceur-ws.org/Vol-202/SEMDESK2006_0014.pdf.

A API Specification

This appendix provides a specification of the operations exposed by the KnowledgeStore API, obtained from the generalization and reorganization of the input collected from project partners. Following the categorization of Section 3.2, Section A.1 defines *intra-layer* operations, while Section A.2 defines *inter-layer* operations. Note that the specification is at a conceptual level, thus abstracting from protocol and format details to be defined in agreement with WP2 system design. Moreover, common concerns such as client authentication and error reporting for malformed requests are not covered: they will be defined as part of the concrete API implementation. Revisions of this specification will be documented in next WP6 deliverables.

A.1 Intra-layer Operations

Intra-layer operations act on data stored in a single layer of the KnowledgeStore, and can be further categorized in (i) *operations on resources representations*, (ii) *CRUD operations* and (iii) *SPARQL access operations*, detailed in the remainder of the section.

A.1.1 Operations on Resources Representations

These operations allow manipulating the representation stored for a resource in the KnowledgeStore. No particular requirement is posed on representations: they are treated as byte sequences and can be anything (e.g., plain text, PDF or RDF/XML NAF files). At most a resource representation can be stored for a resource, and it is optional. Operation `storeResourceRepresentation()` stores the representation of a resource already defined in the system, while operation `retrieveResourceRepresentation()` retrieves it.

HTTP can be used for operations on resource representations. Retrieving a representation can be implemented with the HTTP GET method; storing a representation with the HTTP PUT method (as the resource URI is known). HTTP compression, caching by intermediaries and conditional GET requests (fulfilled only if the requested representation was changed after a previous GET) can be deployed to make the access to resources representations more efficient.

`name`	`storeResourceRepresentation(resource URI, representation)`
`description`	store a resource representation, replacing the currently stored one, if any
`input`	the `URI` of the resource (must exist in the KnowledgeStore); the `representation` of the resource to store
`output`	none on success; an error if the resource URI is unknown
`example`	store the plain text representation of news `nwr:r105`
`notes`	replacing a resource representation is intended as a maintenance task (e.g., to support manual intervention to edit and fix stored data); therefore, resource metadata and depending mentions *are not affected* by the operation: if they become invalid, it is a responsibility of the user to update / delete them

`name`	`retrieveResourceRepresentation(resource URI) : representation`
`description`	retrieve a resource representation, if available
`input`	the `URI` of the resource (must exist in the KnowledgeStore)
`output`	the `representation` stored for the resource if any, otherwise an error signaling either that the URI is unknown or the representation is not available
`example`	retrieve the representation of news `nwr:r105`

A.1.2 CRUD Operations

CRUD operations allow to manipulate sets of objects in a single layer of the KnowledgeStore. They are centred around a limited set of manipulation primitives which are implemented (with some exception and minor customizations) for resources, mentions, entities and statements.[57] No CRUD operations are available for contexts, as they are indirectly specified when manipulating statements and are automatically managed by the system.[58] The remainder of this section presents the CRUD primitives in general, then provides their specifications for the different kinds of manipulable objects.[59]

CRUD primitives CRUD primitives are the usual `create`, `retrieve`, `update` and `delete` primitives, augmented with a `merge` primitive providing an additional way to update data and a `count` primitive for counting matching objects:

`create (object descriptions) : assigned URIs and/or creation errors`
: Stores new objects based on their supplied descriptions. Object URIs are assigned by the system, with the client specifying temporary URIs just for tracking supplied objects. Due to possible data validation, creation may succeed only for a subset of objects, for which URIs are assigned; for the remaining object no data is stored and the corresponding temporary URIs and creation 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 output URIs (or errors) are streamed back to the client.

`retrieve (condition, output attributes) : object descriptions`
: Returns all the objects matching a supplied *condition*. 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 in order to support bulk retrieval operations.

`update (condition, object description, merge criteria) : update errors`
: Updates all the objects matching a supplied condition, setting one or more of their attributes (or entity statements) to a particular value; if the attributes were already set, *merge criteria* (see below) can be optionally used to combine old values with

[57] Both an entity-centric and a statement-centric views of the same entity layer data are offered.

[58] A context is created and its URI assigned when firstly referenced; it is deleted if no more referenced.

[59] As these operations work similarly for resource, mentions, entities and statements, examples are reported only for the mentions case (where they are most useful).

new ones. 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 objects is updated (e.g., because of data validation); for the remaining objects, their URIs and creation errors are reported to the client. It is an error to update non-modifiable attributes (e.g., a statement subject, predicate, object and context).

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

`count (condition) : # matching objects`
: Returns the number of objects matching a supplied condition. The operation is 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.

Merge criteria controls the update of existing data with new one, both for `update` and `merge` primitives. They can be specified either for a mention, resource or statement attribute, in which case they control how its previous values are fused with supplied ones, or for statement predicates, in which case they allow to replace previous statements with new ones for the same predicate (e.g., to replace a statement saying that an entity `foaf:firstName` is 'John' with a new one saying it is 'Johnny'). Supported criteria are:

- `override`, resulting in old data always been replaced by new one (even if the latter is `null`, practically resulting in a deletion);
- `yield`, resulting in new data being set only if no old data exist (i.e., old data takes precedence);
- `union`, resulting in new data values (or statements) being unioned with old ones;

- `min`, resulting in the minimum value among old and new ones being set;[60]
- `max`, resulting in the maximum value among old and new ones being set;

Conditions allow selecting the subset of objects to retrieve or update. They are defined recursively as follows, bu composing basic attribute and statement tests using parenthesis and the AND and OR operators:[61]

```
condition ::=   condition OR condition
                condition AND condition
                ( condition )
                test on attribute value
                test on stmt object { test ... on stmt metadata }
     test ::=   URI = | < | > | <= | >= | != constant
                URI in [constant, constant]
```

CRUD operations on resources All the six `create`, `retrieve`, `update`, `delete`, `merge` and `count` primitives are offered for resources. They operate on *resource descriptions*, which are sets of key-value pairs where the key is an attribute URI and the value is either a scalar (e.g., a string) or a compound value itself described by key-value attribute pairs.

`name`	`createResources(resources) : assigned URIs / creation errors`
`description`	creates one or more resources based on supplied data, generating their URIs
`input`	the `resource descriptions` for the resources to create, with temporary URIs
`output`	mappings from temporary URIs to assigned URIs, or to errors preventing the creation of the corresponding resources

`name`	`mergeResources(resources, merge criteria) : merge errors`
`description`	merges the supplied resource data with data stored in the KnowledgeStore
`input`	`resource descriptions` to be merged, each one supplying a `URI`; `merge criteria` (optional) for (a subset of) the resource attributes
`output`	for each non-merged resource, its URI and the error preventing the merge

`name`	`updateResources(condition, resource, merge crit.) : upd. errors`
`description`	updates the resources satisfying a condition, applying optional merge criteria
`input`	`condition` selecting the resources to update; `resource description` with the attributes to set; `merge criteria` (optional) for (a subset of) the resource attributes
`output`	for each non-updated resource, its URI and the error preventing the update

[60] In case of `min` and `max`, a complete order must be supported by the values datatype.

[61] If applied to a multivalued attribute, a test is satisfied if at least one attribute value satisfies the test.

name	deleteResources(condition) : delete errors
description	deletes all the resources whose attributes satisfy the condition specified
input	condition selecting the resources to delete
output	for each non-deleted resource, its URI and the error preventing the deletion

name	retrieveResources(condition, attribute URIs) : resources
description	retrieves selected attributes of resources satisfying the condition specified
input	condition selecting the resources to retrieve; attribute URIs restricting the output; empty list to select all attributes
output	resource descriptions with requested attributes

name	countResources(condition) : # resources
description	returns the number of resources satisfying the condition specified
input	condition to be matched by resources
output	the number of matching resources

CRUD operations on mentions The six CRUD primitives are offered also for mentions, with mention descriptions defined similarly to resource descriptions.

name	createMentions(mentions) : assigned URIs / creation errors
description	creates one or more mentions based on supplied data, generating their URIs
input	mention descriptions for the mentions to create, with temporary URIs
output	mappings from temporary URIs to assigned URIs, or to errors preventing the creation of the corresponding mentions
example	create two mentions, one with ks:resource = nwr:r103, rdf:type = PER, nif:beginIndex = 4, nif:endIndex = 12, and one with ks:resource = nwr:r103, rdf:type = PER, nif:beginIndex = 131, nif:endIndex = 139; assigned URI are nwr:m501 and nwr:m502

name	mergeMentions(mentions, merge criteria) : merge errors
description	merges the supplied mention data with data stored in the KnowledgeStore
input	mention descriptions to be merged, each one supplying a URI; merge criteria (optional) for (a subset of) the attribute URIs
output	for each non-merged mention, its URI and the error preventing the merge
example	merge local description of mention nwr:m501, having extent = 'John', with its description stored in the KnowledgeStore, having offset = 4; the resulting, stored description has both extent = 'John' and offset = 4

name	updateMentions(condition, mention, merge crit.) : update errors
description	updates the mentions satisfying a condition, applying optional merge criteria
input	condition selecting the mentions to update; mention description with the attributes to set; merge criteria (optional) for (a subset of) the attribute URIs
output	for each non-updated mention, its URI and the error preventing the update
example	clear attribute nwr:polarity of all mentions of type = nwr:EventMention

name	deleteMentions(condition) : delete errors
description	deletes all the mentions whose attributes satisfy the condition specified
input	condition selecting the mentions to delete
output	for each non-deleted mention, its URI and the error preventing the deletion
example	delete all mentions with resource = nwr:r103

name	retrieveMentions(condition, attribute URIs) : mentions
description	retrieves selected attributes of mentions satisfying the condition specified
input	condition selecting the mentions to retrieve; attribute URIs restricting the output; empty list to select all attributes
output	mention descriptions with requested attributes
example	return URI, resource and offset of all the mentions with type = PER

name	countMentions(condition) : # mentions
description	returns the number of mentions satisfying the condition specified
input	condition to be matched by mentions
output	the number of matching mentions

CRUD operations on entities Entities are distinguished by the fact that URIs are assigned externally and their descriptions consist of statements with the entity as subject or object, rather than attributes (a statements can be seen as an attribute extended with context and additional metadata). For these reasons, an extended merge operation, able both to update an entity if the supplied URI is defined in the system, and to create it otherwise, is more convenient for clients than separate create and merge operations, especially when integrating descriptions of the same entity coming from different background knowledge sources. The richer entity description (statements instead of attributes) also allows merge criteria to operate at two levels:

- at the level of statement predicates, merge criteria allow an update operation to possibly replace existing statements for the same entity and RDF predicate;
- at the level of statement metadata, merge criteria define how to merge metadata of old and new statements that happen to have the same subject, predicate, object and context attributes.

As different URIs are usually employed for statement predicates and metadata attributes, a single set of merge criteria is supplied to both `update` and `merge` operations.

`name`	`mergeEntities(entities, merge criteria) : merge errors`
`description`	merges the supplied entity data with data stored in the KnowledgeStore, possibly resulting in the creation of new entities if the supplied URIs are not defined in the system
`input`	`entity descriptions` to be merged/created, with externally assigned URIs; `merge criteria` (optional) for statement predicates or metadata attributes
`output`	for each non-merged/non-created entity, its URI and the causing error

`name`	`updateEntities(condition, entity, merge crit.) : update errors`
`description`	updates the statements of entities satisfying a given condition, applying optional merge criteria
`input`	`condition` selecting the entities to update; `entity description` with the statements to set / modify; `merge criteria` (optional) for statement predicates or metadata attributes
`output`	for each non-updated entity, its URI and the error preventing the update

`name`	`deleteEntities(condition) : delete errors`
`description`	deletes all the entities whose statements satisfy the `condition` specified
`input`	`condition` selecting the entities to delete
`output`	for each non-deleted entity, its URI and the error preventing the deletion

`name`	`retrieveEntities(condition, predicate URIs) : entities`
`description`	retrieves selected statements of entities satisfying the `condition` specified
`input`	`condition` selecting the entities to retrieve; `predicate URIs` restricting the output; empty list selects all statements
`output`	`entity descriptions` with requested statements

`name`	`countEntities(condition) : # entities`
`description`	returns the number of entities satisfying the `condition` specified
`input`	`condition` to be matched by entities
`output`	the number of matching entities

CRUD operations on statements Statements are fine-grained objects that are managed, accordingly to mainstream RDF APIs (e.g., Sesame), using a `merge` semantics only and without an explicit `create` operation. This permits a client to load a bunch of statements with just a `merge` call, which will create the statements if they do not exist or update them (merging their metadata) if they are already defined in the KnowledgeStore.

Apart the different `merge` semantics and the missing `create`, the remaining `update`, `delete`, `retrieve` and `count` primitives are defined for statements too, and operate on a *statement description* which is a set of key-value attribute pairs similarly to resource and mention descriptions.

`name`	`mergeStatements(statements, merge criteria) : merge errors`
`description`	merges the supplied statement data with data stored in the KnowledgeStore, possibly resulting in the creation of new statements
`input`	`statement descriptions` to be merged or created; `merge criteria` (optional) for (a subset of) statement attributes
`output`	for each non-merged/non-created statement, its components and the causing error
`notes`	merge occurs if old and new statements have the same subject, predicate, object and context (which form a statement identity)

`name`	`updateStatements(condition, statement, merge criteria) : update errors`
`description`	updates the statements satisfying a condition, applying optional merge criteria
`input`	`condition` selecting the statements to update; `statement description` with the attributes to set; `merge criteria` (optional) for (a subset of) statement attributes
`output`	for each non-updated statement, its components and the causing error

`name`	`deleteStatements(condition) : delete errors`
`description`	deletes all the statements whose attributes satisfy the `condition` specified
`input`	`condition` selecting the statements to delete
`output`	for each non-deleted statement, its components and the causing error

`name`	`retrieveStatements(condition, attribute URIs) : statements`
`description`	retrieves selected attributes of statements satisfying the `condition` specified
`input`	`condition` selecting the statements to retrieve; `attribute URIs` restricting the output; empty list selects all attributes
`output`	`statement descriptions` with requested attributes

`name`	`countStatements(condition) : # statements`
`description`	returns the number of statements satisfying the `condition` specified
`input`	`condition` to be matched by statements
`output`	the number of matching statements

A.1.3 SPARQL Access to Statements

SPARQL query access complying with the SPARQL Protocol W3C standard [Feigenbaum *et al.*, 2013] is offered for indexed statements (i.e., the ones in the triple store, see Section 4.2.2), and consists in the possibility of evaluating SPARQL SELECT, ASK, CONSTRUCT and DESCRIBE queries over all or a subset of indexed statements as identified by a SPARQL *dataset* [Seaborne and Harris, 2013]. A dataset consists of two sets of graph URIs: graphs in the first sets are merged together forming the default graph for the query; graphs in the second set become the named graphs for the query. The dataset is optionally supplied by the client, either as an operation parameter or embedded in the query string (`FROM` and `FROM NAMED` clauses). If not specified, a default dataset is used, having all the stored graphs as named graphs and their RDF merge as the default graph.

SPARQL query results will be computed considering all the statements that can be inferred according to the logical inference solution to be defined in T6.3. From a conceptual point of view (the implementation may be different), the approach can be assimilated to a two-steps procedure: (i) all the inferences are materialized; and (ii) the SPARQL operation is evaluated over all the resulting statements, being them explicitly stated or inferred. It is worth clarifying that this does not correspond to the adoption of a (non-*simple*) SPARQL *entailment regime* [Glimm and Ogbuji, 2013], as it would allow deriving different sets of inferences according to the dataset of the client query [Glimm and Ogbuji, 2013, Section 9], thus preventing techniques such as (partial) closure materialization and hence scalability.

`name`	`sparqlQuery(query, dataset) : query solutions or triples`
`description`	evaluates the supplied SPARQL `query` on indexed statements or a subset of them identified by the `dataset` parameter
`input`	the `query` string, in the SELECT, ASK, CONSTRUCT or DESCRIBE forms; an optional `dataset` specification
`output`	on success, 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

`example`	evaluate the following query:
	`SELECT ?p ?e FROM nwr:ctx106 WHERE {`
	`  ?p a foaf:Person .`
	`  ?e a nwr:SellEvent ; sem:hasActor ?p }`

`notes`	results are streamed to the client

A.2 Inter-layer Operations

Inter-layer operations comprise the general-purpose `match()` operation as well as a number of NewsReader-specific operations collected from project partners.

name	match(condition and output attribute URIs at resource, mention and entity level) : matching <resource, mention, entity> tuples
description	returns a set of ⟨resource, mention, entity⟩ tuples whose mention occurs in the resource and refers to the entity, and such that their attributes and statements satisfy a certain condition; for each tuple, a specified set of output attributes for each resource, mention and entity element is returned
input	a condition over the attributes of matched resources and mentions and over the statements of matched entities; a list of predicate URIs identifying output attributes for resources and mentions, and output statements for entities
output	an unordered list of matching ⟨resource, mention, entity⟩ tuples with the output attributes specified
example	retrieve uri and dc:title of all the resources of type nwr:News in which entity dbpedia:Berlin is mentioned
notes	results are streamed to the client

name	getLocationsByInstance(entity URI, time period)
description	gives a list of LOC entities where an instance of an entity is mentioned
input	the ID for the instance (either event or entity)
output	list with the instance ID
example	give a list of location instances in which Merkel was located in the period 2011 till 2012
notes	possibly restricted by periods

name	getEventMentionsByRoleByEntityType(role, entity type)
description	lists all the mentions of events for an entity type with the specified role and the attributes of the events to return (e.g. type)
input	the type of Role and the type of entity
output	a list with the mention ID of the events and the selected attributes for each matching event mention
example	provide a list of event mentions in which PER+CEO has the A2 (Propbank) role in the period 2005 till 2012, attributes are event type, date of publication

name	getParticipantMentionsByRoleByEventType(role, entity type)
description	lists all the mentions of entities participating with the specified role for an event type and the attributes of the events to return (e.g. type)
input	the type of Role and the type of event
output	a list with the mention ID of the participants and the selected attributes for each matching participant mention
example	list all mentions of entities that have the A2 role in events of the type EVENT+Fired, attributes: name, type, date, location

B Core Data Model Ontology

This appendix specifies the *KnowledgeStore Core Data Model Ontology*, an OWL 2 ontology that formalizes the core concepts of the KnowledgeStore data model. In addition, it declares a number of annotation properties for specifying how instance data has to be stored in a KnowledgeStore instance. These properties currently contain only the ks:storedAttribute annotation property, which specifies which attributes are stored in the KnowledgeStore for the instances of a certain class; additional properties will be defined as part of the implementation in order to control the generation of keys, replication and the deployment of additional indexes. The ontology is reported in the following listing using the the Manchester Syntax, with `rdfs:comment`s documenting its main concepts. Following best practices for ontology publishing, it is also available (also in additional syntaxes, accessible via content negotiation) at the URL `http://dkm.fbk.eu/ontologies/knowledgestore`.

```
Prefix: dc:   <http://purl.org/dc/terms/>
Prefix: nie:  <http://www.semanticdesktop.org/ontologies/2007/01/19/nie#>
Prefix: nfo:  <http://www.semanticdesktop.org/ontologies/2007/03/22/nfo#>
Prefix: ks:   <http://dkm.fbk.eu/ontologies/knowledgestore#>

# ONTOLOGY DECLARATION

Ontology: <http://dkm.fbk.eu/ontologies/knowledgestore>
Import: <http://purl.org/dc/terms/>
Import: <http://www.semanticdesktop.org/ontologies/2007/01/19/nie>
Import: <http://www.semanticdesktop.org/ontologies/2007/03/22/nfo>
Annotations: rdfs:label "KnowledgeStore Core Data Model ontology"@en,
             rdfs:comment "This ontology formalizes the core concepts of the KnowledgeStore data model, i.e.,
                the classes, attributes and relations of the fixed part of the model (i.e., the TBox); these
                concepts can be extended by derived ontologies in order to configure specific KnowledgeStore
                instances. This ontology provides also a number of annotation properties for specifying how
                instance data (i.e., the ABox) has to be stored by the KnowledgeStore."@en

# IMPORTED TERMS (REQUIRED BY PROTEGE)

Class: rdf:Property                 Class: rdfs:Class                 Datatype: rdfs:Literal
DataProperty: nfo:fileName          DataProperty: nfo:fileSize        DataProperty: nfo:fileCreated
DataProperty: nfo:fileLastModified  ObjectProperty: dc:format         ObjectProperty: nie:isStoredAs

# ANNOTATION PROPERTIES

AnnotationProperty: ks:storedAttribute
Domain: <http://www.w3.org/2002/07/owl#Class> Range: <http://www.w3.org/1999/02/22-rdf-syntax-ns#
    Property>
Annotations: rdfs:label "stored attribute"@en,
             rdfs:comment "Specifies which attributes are stored in the KnowledgeStore for instances of a certain
                class; note that attributes stored for a super-class are inherited by its sub-classes."@en

# RESOURCES

Class: ks:Resource
Annotations: ks:storedAttribute nie:isStoredAs,        ks:storedAttribute ks:contains,
             rdfs:label "resource"@en,
             rdfs:comment "A self-contained and identifiable information object with a digital representation,
                such as a news article, a photo or a video."@en

Class: nfo:FileDataObject
Annotations: ks:storedAttribute dc:format,             ks:storedAttribute nfo:fileName,
             ks:storedAttribute nfo:fileSize,          ks:storedAttribute nfo:fileCreated,
```

```
                ks:storedAttribute  nfo:fileLastModified ,
                rdfs:label  "file "@en,
                rdfs:comment "A digital file holding the representaiton of a resource. Instances of this class are
                    automatically managed by the KnowledgeStore based on uploaded resource representations "@en

ObjectProperty: ks:contains
Domain: ks:Resource Range: ks:Mention
Annotations: rdfs:label  "contains"@en,
                rdfs:comment "Denotes the mentions contained in a resource."@en

# MENTIONS

Class: ks:Mention
SubClassOf: ks:containedIn exactly 1 ks:Resource
Annotations: ks:storedAttribute ks:containedIn , ks:storedAttribute ks:refersTo ,
                rdfs:label  "mention"@en,
                rdfs:comment "A fragment of resource (e.g., a piece of text or a bunch of pixels ) that denotes
                    something of interest , such as an entity or a relation among entity, a concept."@en

ObjectProperty: ks:containedIn
Characteristics: Functional InverseOf: ks:contains

ObjectProperty: ks:refersTo
Domain: ks:Mention Range: ks:Entity Characteristics: Functional
Annotations: rdfs:label  "refers to"@en,
                rdfs:comment "Denotes the entity a given mention refers to."@en

# ENTITIES

Class: ks:Entity
Annotations: ks:storedAttribute ks:referredBy , # denotes the entity mentions
                rdfs:label  "entity "@en,
                rdfs:comment "Any identifiable entity in the domain of discourse , extracted from text and/or
                    imported from some source of background knowledge."@en

ObjectProperty: ks:referredBy
InverseOf: ks:refersTo

# STATEMENTS

Class: ks:Statement # not closed to type, attribute and relation cases to allow for more complex statements
SubClassOf: ks:predicate some rdf:Property, ks:context some ks:Context
Annotations: ks:storedAttribute ks:subject , ks:storedAttribute ks:predicate ,
                ks:storedAttribute ks:context , ks:storedAttribute ks:extractedFrom,
                rdfs:label  "statement"@en,
                rdfs:comment "A <subject, predicate, object> triple possibly extracted from some mentions and
                    holding in a specific context, that describes some feature of a subject entity ."@en

ObjectProperty: ks:subject
Domain: ks:Statement Range: ks:Entity Characteristics: Functional
Annotations: rdfs:label  "subject"@en,
                rdfs:comment "Denotes the subject of a Statement."@en

ObjectProperty: ks:predicate
Domain: ks:Statement Range: rdf:Property Characteristics: Functional
Annotations: rdfs:label  "predicate "@en,
                rdfs:comment "Denotes the predicate of a Statement."@en

ObjectProperty: ks:context
Domain: ks:Statement Range: ks:Context Characteristics: Functional
Annotations: rdfs:label  "context"@en,
                rdfs:comment "Denotes the context a statement holds in"@en

ObjectProperty: ks:extractedFrom
Domain: ks:Statement Range: ks:Mention
```

```
109 Annotations: rdfs:label "extracted from"@en,
110              rdfs:comment "Denotes the mention a statement has been extracted from."@en
111
112 # KINDS OF STATEMENTS
113
114 Class: ks:TypeStatement
115 EquivalentTo: ks:Statement and ks:type some owl:Thing
116 Annotations: ks:storedAttribute ks:type ,
117              rdfs:label "type statement"@en,
118              rdfs:comment "A statement expressing the type of an entity ."@en
119
120 ObjectProperty: ks:type
121 Domain: ks:TypeStatement Range: rdfs:Class Characteristics: Functional
122 Annotations: rdfs:label "type"@en,
123              rdfs:comment "Denotes the type of a TypeStament."@en
124
125 Class: ks:AttributeStatement
126 EquivalentTo: ks:Statement and ks:value exactly 1 rdfs:Literal
127 Annotations: ks:storedAttribute ks:value ,
128              rdfs:label "attribute statement"@en,
129              rdfs:comment "A statement expressing an attribute of an entity ."@en
130
131 DataProperty: ks:value
132 Domain: ks:AttributeStatement Range: rdfs:Literal Characteristics: Functional
133 Annotations: rdfs:label "value"@en,
134              rdfs:comment "Denotes the value of an AttributeStatement."@en
135
136 Class: ks:RelationStatement
137 EquivalentTo: ks:Statement and ks:object some ks:Entity
138 Annotations: ks:storedAttribute ks:object ,
139              rdfs:label "relation statement"@en,
140              rdfs:comment "A statement expressing a relation among two entities. The first entity is denoted by
141                  the subject, the second one by mandatory attribute hasObject."@en
142
143 ObjectProperty: ks:object
144 Domain: ks:RelationStatement Range: ks:Entity Characteristics: Functional
145 Annotations: rdfs:label "object"@en,
146              rdfs:comment "Denotes the object of a RelationStatement."@en
147
148 # CONTEXTS
149
150 Class: ks:Context
151 Annotations: rdfs:label "context"@en,
152              rdfs:comment "A region in a space of contextual dimensions (e.g., time, point of view) where
153                  certain statements hold, identified by a URI."@en
154
155 # DISJOINTNESS CONSTRAINTS
156
157 DisjointClasses: ks:Resource, ks:Mention, ks:Entity , ks:Statement, ks:Context, nfo:FileDataObject
158 DisjointClasses: ks:TypeStatement, ks:AttributeStatement , ks:RelationStatement
```

C NewsReader Data Model Ontology

This appendix specifies the NewsReader *Data Model Ontology*, an OWL 2 ontology that formalizes the data model of the KnowledgeStore instance for NewsReader through the specialization of the Core Data Model Ontology. The ontology is reported in the following listing using the the Manchester Syntax, with `rdfs:comment`s documenting its main concepts. Following best practices for ontology publishing, it is also available (also in additional syntaxes, accessible via content negotiation) at the URL `http://dkm.fbk.eu/ontologies/newsreader`.

```
Prefix: dc:   <http://purl.org/dc/terms/>
Prefix: nif:  <http://nlp2rdf.lod2.eu/schema/string/>
Prefix: sem:  <http://semanticweb.cs.vu.nl/2009/11/sem/>
Prefix: nie:  <http://www.semanticdesktop.org/ontologies/2007/01/19/nie#>
Prefix: nfo:  <http://www.semanticdesktop.org/ontologies/2007/03/22/nfo#>
Prefix: ks:   <http://dkm.fbk.eu/ontologies/knowledgestore#>
Prefix: nwr:  <http://dkm.fbk.eu/ontologies/newsreader#>

# ONTOLOGY DECLARATION

Ontology: <http://dkm.fbk.eu/ontologies/newsreader>
Import: <http://dkm.fbk.eu/ontologies/knowledgestore>
Import: <http://semanticweb.cs.vu.nl/2009/11/sem/>
Import: <http://nlp2rdf.lod2.eu/schema/string/>
Import: <http://www.semanticdesktop.org/ontologies/2007/03/22/nfo>
Annotations: rdfs:label "NewsReader Data Model ontology"@en,
             rdfs:comment "This ontology defines the data model of the NewsReader KnowledgeStore instance."@en

# IMPORTED TERMS (REQUIRED BY PROTEGE)

Class: ks:Entity                    Class: nfo:FileDataObject           Class: nfo:TextDocument
Class: dc:PeriodOfTime              DataProperty: nie:keyword           DataProperty: nif:beginIndex
DataProperty: nif:endIndex          DataProperty: nif:anchorOf          DataProperty: sem:hasEndTimeStamp
DataProperty: dc:title              DataProperty: dc:issued             DataProperty: sem:hasBeginTimeStamp
DataProperty: nfo:characterCount    DataProperty: nfo:wordCount         AnnotationProperty: rdfs:comment
AnnotationProperty: dc:source       AnnotationProperty: dc:subject      AnnotationProperty: ks:storedAttribute
ObjectProperty: dc:publisher        ObjectProperty: dc:contributor      ObjectProperty: dc:creator
ObjectProperty: dc:spatial          ObjectProperty: dc:temporal         ObjectProperty: dc:rights
ObjectProperty: dc:rightsHolder     ObjectProperty: dc:language         ObjectProperty: nfo:fileUrl
ObjectProperty: ks:refersTo         ObjectProperty: sem:accordingTo

# RESOURCES

Class: ks:Resource
Annotations: ks:storedAttribute <http://www.w3.org/2000/01/rdf-schema#comment>

Class: nwr:NAFAnnotation
SubClassOf: ks:Resource, nwr:annotationOf some nwr:News, nfo:TextDocument
Annotations: ks:storedAttribute nwr:layer ,
             ks:storedAttribute nwr:annotationOf,
             rdfs:label "NAF annotation"@en,
             rdfs:comment "The annotation of a news according to the NAF format, consisting in one or more layers
                 of NLP annotations encoded in a standoff, XML-based format."@en

Class: nwr:News
SubClassOf: ks:Resource, nfo:TextDocument
Annotations: ks:storedAttribute dc:title ,          ks:storedAttribute dc:publisher ,
             ks:storedAttribute dc:creator ,        ks:storedAttribute dc:issued ,
             ks:storedAttribute dc:spatial ,        ks:storedAttribute dc:temporal,
             ks:storedAttribute dc:rights ,         ks:storedAttribute dc:rightsHolder ,
```

```
                ks:storedAttribute  dc:language,          ks:storedAttribute  nie:keyword,
                ks:storedAttribute  nfo:fileUrl ,         ks:storedAttribute  nfo:characterCount,
                ks:storedAttribute  nfo:wordCount,        ks:storedAttribute  nwr:annotatedWith,
                ks:storedAttribute  <http://purl.org/dc/terms/subject>,
                rdfs:label  "news"@en,
                rdfs:comment "A news article, consisting in the news plain text and associated metadata."@en

ObjectProperty: nwr:layer
Domain: nwr:NAFAnnotation Range: nwr:NAFLayer
Annotations: rdfs:label  "available layer "@en,
                rdfs:comment "Specifies the NAF layers available in a NAF annotation resource"@en

Class: nwr:NAFLayer
EquivalentTo: { nwr:naf_text , nwr:naf_terms, nwr:naf_deps, nwr:naf_chunks, nwr:naf_entities ,
                  nwr:naf_coreferences , nwr:naf_opinions , nwr:naf_events , nwr:naf_timex3 }
Annotations: rdfs:label  "NAF layer"@en,
                rdfs:comment "A NAF layers. Currently defined layers include text, terms, dependencies (deps), chunks,
                  entities , coreferences , opinions , events and timex3 expressions ."@en

Individual:  nwr:naf_text       Individual:  nwr:naf_terms      Individual:  nwr:naf_deps
Individual:  nwr:naf_chunks     Individual:  nwr:naf_entities   Individual:  nwr:naf_coreferences
Individual:  nwr:naf_opinions   Individual:  nwr:naf_events     Individual:  nwr:naf_timex3

ObjectProperty: nwr:annotationOf
Domain: nwr:NAFAnnotation Range: nwr:News Characteristics: Functional
Annotations: rdfs:label  "annotation of"@en,
                rdfs:comment "Specifies the news resource a NAF annotation resource is associated to."@en

ObjectProperty: nwr:annotatedWith
InverseOf: nwr:annotationOf
Annotations: rdfs:label  "annotated with"@en,
                rdfs:comment "Specifies the NAF annotation(s) associated to a news resource."@en

DataProperty: nwr:beginTime
Domain: dc:PeriodOfTime Range: xsd:date
Annotations: rdfs:label  "begin time"@en,
                rdfs:comment "The begin time of a period of time (may be missing, denoting an open ended period)."@en

DataProperty: nwr:endTime
Domain: dc:PeriodOfTime Range: xsd:date
Annotations: rdfs:label  "end time"@en,
                rdfs:comment "The end time of a period of time (may be missing, denoting an open ended period)."@en

# MENTIONS CLASS HIERARCHY

Class: ks:Mention # nif:beginIndex , nif:endIndex , nif:anchorOf mandatory for all mentions
SubClassOf: nif:beginIndex some xsd:integer, nif:endIndex some xsd:integer, nif:anchorOf some xsd:string
Annotations: ks:storedAttribute  nif:beginIndex ,
                ks:storedAttribute  nif:endIndex ,
                ks:storedAttribute  nif:anchorOf ,
                ks:storedAttribute  nwr:confidence,
                ks:storedAttribute  <http://www.w3.org/2000/01/rdf-schema#comment>

Class: nwr:SignalMention
SubClassOf: ks:Mention, ks:refersTo max 0 owl:Thing
Annotations: rdfs:label  "signal mention"@en,
                rdfs:comment "A piece of text supporting the existence of a causal (CLink) or temporal (TLink)
                  relation among events and/or time expressions ."@en

Class: nwr:ValueMention
SubClassOf: ks:Mention, ks:refersTo max 0 owl:Thing
Annotations: ks:storedAttribute  nwr:valueType,
                rdfs:label  "value mention"@en,
                rdfs:comment "A numerical expression denoting either a quantity (cardinal numbers in general), a
                  percentage or a monetary value."@en
```

```
116
117 Class: nwr:EntityMention
118 SubClassOf: ks:Mention
119 Annotations: rdfs:label "entity mention"@en,
120                rdfs:comment "A piece of text denoting an entity in the domain of discourse (identified by relation
121                    nwr:refersTo), such as a person, organization or location."@en
122
123 Class: nwr:RelationMention
124 SubClassOf: ks:Mention, ks:refersTo max 0 owl:Thing
125 Annotations: rdfs:label "relation mention"@en,
126                rdfs:comment "A piece of text expressign a relation between two entities, whose mentions are
127                    identified by nwr:arg1 and nwr:arg2 links)."@en
128
129 Class: nwr:ObjectMention
130 SubClassOf: nwr:EntityMention
131 Annotations: ks:storedAttribute nwr:head, ks:storedAttribute nwr:syntacticType,
132                ks:storedAttribute nwr:referenceType, ks:storedAttribute nwr:entityType,
133                rdfs:label "object mention"@en,
134                rdfs:comment "A mention of an endurant object (in KR literature), such as a person, organization or
135                    location (known as 'entities' in the NLP literature)."@en
136
137 Class: nwr:TimeOrEventMention
138 SubClassOf: nwr:EntityMention
139 EquivalentTo: nwr:EventMention or nwr:TimeMention
140 Annotations: rdfs:label "time or event mention"@en,
141                rdfs:comment "Utility concept aggregating mentions of events and mentions of time expressions."@en
142
143 Class: nwr:EventMention
144 SubClassOf: nwr:TimeOrEventMention
145 Annotations: ks:storedAttribute nwr:pred, ks:storedAttribute nwr:pos,
146                ks:storedAttribute nwr:factual, ks:storedAttribute nwr:tense,
147                ks:storedAttribute nwr:aspect, ks:storedAttribute nwr:vform,
148                ks:storedAttribute nwr:polarity, ks:storedAttribute nwr:mood,
149                ks:storedAttribute nwr:modality,
150                rdfs:label "event mention"@en,
151                rdfs:comment "A mention of an event."@en
152
153 Class: nwr:TimeMention
154 SubClassOf: nwr:TimeOrEventMention
155 Annotations: ks:storedAttribute nwr:value,
156                ks:storedAttribute nwr:timeType,
157                rdfs:label "time mention"@en,
158                rdfs:comment "A mention of a time expression."@en
159
160 Class: nwr:TLink
161 SubClassOf: nwr:RelationMention, nwr:arg1 some nwr:TimeOrEventMention, nwr:arg2 some nwr:TimeOrEventMention
162 Annotations: ks:storedAttribute nwr:relType,
163                ks:storedAttribute nwr:signal,
164                rdfs:label "TLink"@en,
165                rdfs:comment "A temporal link, i.e., a mention denoting a temporal relation among two events and/or
166                    time expressions."@en
167
168 Class: nwr:CLink
169 SubClassOf: nwr:RelationMention, nwr:arg1 some nwr:EventMention, nwr:arg2 some nwr:EventMention
170 Annotations: ks:storedAttribute nwr:signal,
171                rdfs:label "CLink"@en,
172                rdfs:comment "A causal link, i.e., a mention denoting a causal relation among two events."@en
173
174 Class: nwr:SLink
175 SubClassOf: nwr:RelationMention, nwr:arg1 some nwr:EventMention, nwr:arg2 some nwr:EventMention
176 Annotations: rdfs:label "SLink"@en,
177                rdfs:comment "A structural link, i.e., a mention denoting a structural relation among two events."@en
178
179 Class: nwr:Participation
180 SubClassOf: nwr:RelationMention, nwr:arg1 some nwr:EventMention, nwr:arg2 some nwr:ObjectMention
```

```
Annotations: ks:storedAttribute nwr:semRole,
             ks:storedAttribute nwr:dep,
             rdfs:label "participation "@en,
             rdfs:comment "A mention denoting the participation of an object (e.g., a person) to a certain event,
                further characterized by the role played by that object and a syntactic dependency among the
                    object
                and the event."@en

# ATTRIBUTES OF VALUE MENTIONS

ObjectProperty: nwr:valueType
Domain: nwr:ValueMention Range: nwr:ValueType Characteristics: Functional
Annotations: rdfs:label "value type"@en,
             rdfs:comment "Specifies the type of value expressed by a value mention."@en

Class: nwr:ValueType
EquivalentTo: { nwr:value_percent , nwr:value_money, nwr:value_quantity }
Annotations: rdfs:label "value type"@en,
             rdfs:comment "Enumeration of value types: either a percentage (nwr:value_percent ), a monetary value
                (nwr:value_money) or a generic quantity (nwr:value_quantity )."@en

Individual: nwr:value_percent Individual: nwr:value_money Individual: nwr:value_quantity

# ATTRIBUTES OF OBJECT MENTIONS

DataProperty: nwr:head
Domain: nwr:ObjectMention Range: xsd:string Characteristics: Functional
Annotations: rdfs:label "head"@en,
             rdfs:comment "Specifies the head of a mention, which is a string contained in the mention extent."@en

ObjectProperty: nwr:syntacticType
Domain: nwr:ObjectMention Range: nwr:SyntacticType Characteristics: Functional
Annotations: rdfs:label "syntactic type"@en,
             rdfs:comment "Specifies the syntactic category of the mention."@en

Class: nwr:SyntacticType
EquivalentTo: { nwr:syn_nam, nwr:syn_nom, nwr:syn_pro, nwr:syn_whq, nwr:syn_ptv,
                nwr:syn_app, nwr:syn_conj , nwr:syn_pre , nwr:syn_other }
Annotations: rdfs:label "syntactic type"@en,
             rdfs:comment "Enumeration of syntactic types, such as proper name (nwr:syn_nam), pronoun
                (nwr:syn_pro), ... "@en

Individual: nwr:syn_nam Individual: nwr:syn_nom Individual: nwr:syn_pro
Individual: nwr:syn_whq Individual: nwr:syn_ptv Individual: nwr:syn_app
Individual: nwr:syn_conj Individual: nwr:syn_pre Individual: nwr:syn_other

ObjectProperty: nwr:referenceType
Domain: nwr:ObjectMention Range: nwr:ReferenceType Characteristics: Functional
Annotations: rdfs:label "reference type"@en,
             rdfs:comment "Specifies the kind of reference a mention makes to the entity ."@en

Class: nwr:ReferenceType
EquivalentTo: { nwr:ref_spc , nwr:ref_gen , nwr:ref_usp , nwr:ref_neg }
Annotations: rdfs:label "reference type"@en,
             rdfs:comment "Enumeration of reference types. Possible values are: nwr:ref_spc (specific referential ),
                nwr:ref_gen (generic referential ), nwr:ref_usp (under-specified referential ), nwr:ref_neg
                (negatively quantified )."@en

Individual: nwr:ref_spc Individual: nwr:ref_gen
Individual: nwr:ref_usp Individual: nwr:ref_neg

ObjectProperty: nwr:entityType
Domain: nwr:ObjectMention Range: nwr:EntityType Characteristics: Functional
Annotations: rdfs:label "entity type"@en,
             rdfs:comment "Specifies the semantic type of the mentioned entity ."@en
```

```
245
246 Class: nwr:EntityType
247 EquivalentTo: { nwr:ent_person, nwr:ent_location , nwr:ent_organization , nwr:ent_artifact , nwr:ent_financial }
248 Annotations: rdfs:label "entity type"@en,
249              rdfs:comment "Enumeration of entity types."@en
250
251 Individual: nwr:ent_person  Individual: nwr:ent_location  Individual: nwr:ent_organization
252 Individual: nwr:ent_artifact  Individual: nwr:ent_financial
253
254 # ATTRIBUTES OF EVENT MENTIONS
255
256 ObjectProperty: nwr:eventType
257 Domain: nwr:EventMention Range: nwr:EventType Characteristics: Functional
258 Annotations: rdfs:label "event type"@en,
259              rdfs:comment "Specifies the semantic type of the mentioned event."@en
260
261 Class: nwr:EventType
262 EquivalentTo: { nwr:ev_speech_cognitive , nwr:ev_contextual , nwr:ev_grammatical }
263 Annotations: rdfs:label "event type"@en,
264              rdfs:comment "Enumeration of event types."@en
265
266 Individual: nwr:ev_speech_cognitive  Individual: nwr:ev_contextual  Individual: nwr:ev_grammatical
267
268 DataProperty: nwr:pred
269 Domain: nwr:EventMention Range: xsd:string Characteristics: Functional
270 Annotations: rdfs:label "pred"@en,
271              rdfs:comment "Specifies the lemma of the token describing the event."@en
272
273 ObjectProperty: nwr:pos
274 Domain: nwr:EventMention Range: nwr:PartOfSpeech Characteristics: Functional
275 Annotations: rdfs:label "pos"@en,
276              rdfs:comment "Specifies the part-of-speech for the event mention."@en
277
278 Class: nwr:PartOfSpeech
279 EquivalentTo: { nwr:pos_adjective , nwr:pos_noun, nwr:pos_verb, nwr:pos_preposition , nwr:pos_other }
280 Annotations: rdfs:label "part-of-speech"@en,
281              rdfs:comment "Enumeration of possible part-of-speech."@en
282
283 Individual: nwr:pos_adjective  Individual: nwr:pos_noun  Individual: nwr:pos_verb
284 Individual: nwr:pos_preposition  Individual: nwr:pos_other
285
286 DataProperty: nwr:factual
287 Domain: nwr:EventMention Range: xsd:boolean Characteristics: Functional
288 Annotations: rdfs:label "factual "@en,
289              rdfs:comment "Specifies whether the mentioned event is factual ."@en
290
291 ObjectProperty: nwr:tense
292 Domain: nwr:EventMention Range: nwr:Tense Characteristics: Functional
293 Annotations: rdfs:label "tense"@en,
294              rdfs:comment "Specifies the tense of the verb conveying the mentioned event."@en
295
296 Class: nwr:Tense
297 EquivalentTo: { nwr:tense_future , nwr:tense_past , nwr:tense_present , nwr:tense_infinitive ,
298                 nwr:tense_prespart , nwr:tense_pastpart , nwr:tense_none }
299 Annotations: rdfs:label "tense"@en,
300              rdfs:comment "Enumeration of verb tenses."@en
301
302 Individual: nwr:tense_future  Individual: nwr:tense_past  Individual: nwr:tense_present
303 Individual: nwr:tense_infinitive  Individual: nwr:tense_prespart  Individual: nwr:tense_pastpart
304 Individual: nwr:tense_none
305
306 ObjectProperty: nwr:aspect
307 Domain: nwr:EventMention Range: nwr:Aspect Characteristics: Functional
308 Annotations: rdfs:label "aspect"@en,
309              rdfs:comment "Specifies the aspect of the verb conveying the mentioned event."@en
```

```
310
311 Class: nwr:Aspect
312 EquivalentTo: { nwr:aspect_progressive , nwr:aspect_perfective , nwr:aspect_imperfective ,
313                  nwr:aspect_perfective_progressive , nwr:aspect_imperfective_progressive , nwr:aspect_none }
314 Annotations: rdfs:label "aspect"@en,
315              rdfs:comment "Enumeration of verb aspects."@en
316
317 Individual: nwr:aspect_progressive          Individual: nwr:aspect_perfective
318 Individual: nwr:aspect_imperfective         Individual: nwr:aspect_perfective_progressive
319 Individual: nwr:aspect_imperfective_progressive  Individual: nwr:aspect_none
320
321 ObjectProperty: nwr:vform
322 Domain: nwr:EventMention Range: nwr:VerbForm Characteristics: Functional
323 Annotations: rdfs:label "vform"@en,
324              rdfs:comment "Specifies the form of the verb conveying the mentioned event."@en
325
326 Class: nwr:VerbForm
327 EquivalentTo: { nwr:vform_infinitive , nwr:vform_gerund, nwr:vform_participle , nwr:vform_none }
328 Annotations: rdfs:label "verb form"@en,
329              rdfs:comment "Enumeration of verb forms."@en
330
331 Individual: nwr:vform_infinitive  Individual: nwr:vform_gerund
332 Individual: nwr:vform_participle  Individual: nwr:vform_none
333
334 ObjectProperty: nwr:polarity
335 Domain: nwr:EventMention Range: nwr:Polarity Characteristics: Functional
336 Annotations: rdfs:label "polarity "@en,
337              rdfs:comment "Specifies the polarity of the mentioned event."@en
338
339 Class: nwr:Polarity
340 EquivalentTo: { nwr:polarity_pos , nwr:polarity_neg }
341 Annotations: rdfs:label "polarity "@en,
342              rdfs:comment "Enumeration of event polarities (either positive or negative)."@en
343
344 Individual: nwr:polarity_pos  Individual: nwr:polarity_neg
345
346 ObjectProperty: nwr:mood
347 Domain: nwr:EventMention Range: nwr:Mood Characteristics: Functional
348 Annotations: rdfs:label "mood"@en,
349              rdfs:comment "Specifies the mood of the verb conveying the mentioned event."@en
350
351 Class: nwr:Mood
352 EquivalentTo: { nwr:mood_indicative, nwr:mood_conditional, nwr:mood_subjunctive,
353                  nwr:mood_imperative, nwr:mood_none }
354 Annotations: rdfs:label "mood"@en,
355              rdfs:comment "Enumeration of verb moods."@en
356
357 Individual: nwr:mood_indicative  Individual: nwr:mood_conditional  Individual: nwr:mood_subjunctive
358 Individual: nwr:mood_imperative  Individual: nwr:mood_none
359
360 DataProperty: nwr:modality
361 Domain: nwr:EventMention Range: xsd:string Characteristics: Functional
362 Annotations: rdfs:label "modality"@en,
363              rdfs:comment "Conveys different degrees of modality of an event. Its value is the lemma of the modal
364                  verb modifying the main event, e.g., may (English), potere (Italian ), poder (Spanish)."@en
365
366 # ATTRIBUTES OF TIME MENTIONS
367
368 DataProperty: nwr:value
369 Domain: nwr:TimeMention Range: xsd:string Characteristics: Functional
370 Annotations: rdfs:label "value"@en,
371              rdfs:comment "Specifies the normalized value of a temporal expression using the ISO-8601 standard."
                     @en
372
373 ObjectProperty: nwr:timeType
```

```
374 Domain: nwr:TimeMention Range: nwr:TIMEX3Type Characteristics: Functional
375 Annotations: rdfs:label "time type"@en,
376              rdfs:comment "Specifies the type of time expressed by a time mention."@en
377
378 Class: nwr:TIMEX3Type
379 EquivalentTo: { nwr:timex3_date, nwr:timex3_time, nwr:timex3_duration, nwr:timex3_set }
380 Annotations: rdfs:label "TIMEX3 type"@en,
381              rdfs:comment "Enumeration of TIMEX3 temporal expression types."@en
382
383 Individual: nwr:timex3_date      Individual: nwr:timex3_time
384 Individual: nwr:timex3_duration  Individual: nwr:timex3_set
385
386 # ATTRIBUTES OF RELATION MENTIONS
387
388 ObjectProperty: nwr:arg1
389 Domain: nwr:RelationMention Range: nwr:EntityMention Characteristics: Functional
390 Annotations: rdfs:label "arg1"@en,
391              rdfs:comment "Specifies the first argument of a relation mention."@en
392
393 ObjectProperty: nwr:arg2
394 Domain: nwr:RelationMention Range: nwr:EntityMention Characteristics: Functional
395 Annotations: rdfs:label "arg2"@en,
396              rdfs:comment "Specifies the second argument of a relation mention."@en
397
398 ObjectProperty: nwr:signal
399 Domain: nwr:RelationMention Range: nwr:SignalMention Characteristics: Functional
400 Annotations: rdfs:label "signal "@en,
401              rdfs:comment "Associates a relation mention to the signal mention denoting the existence of the
402                  relation ."@en
403
404 ObjectProperty: nwr:relType
405 Domain: nwr:TLink Range: nwr:TLinkType Characteristics: Functional
406 Annotations: rdfs:label "relation type"@en,
407              rdfs:comment "Specifies the type of TLink relation ."@en
408
409 Class: nwr:TLinkType
410 EquivalentTo: { nwr:rel_before , nwr:rel_after , nwr:rel_includes , nwr:rel_is_included , nwr:rel_simultaneous ,
411                 nwr:rel_iafter , nwr:rel_ibefore , nwr:rel_begins , nwr:rel_ends , nwr:rel_begun_by ,
412                 nwr:rel_ended_by , nwr:rel_measure }
413 Annotations: rdfs:label "TLink type"@en,
414              rdfs:comment "Enumeration of TLink types."@en
415
416 Individual: nwr:rel_before      Individual: nwr:rel_after       Individual: nwr:rel_includes
417 Individual: nwr:rel_is_included Individual: nwr:rel_simultaneous Individual: nwr:rel_iafter
418 Individual: nwr:rel_ibefore    Individual: nwr:rel_begins      Individual: nwr:rel_ends
419 Individual: nwr:rel_begun_by   Individual: nwr:rel_ended_by    Individual: nwr:rel_measure
420
421 ObjectProperty: nwr:semRole
422 Domain: nwr:Participation Range: nwr:Role Characteristics: Functional
423 Annotations: rdfs:label "semantic role "@en,
424              rdfs:comment "Specifies the semantic role played by thje object in the scope of a participation
425                  mention."@en
426
427 Class: nwr:Role # roles taken from FrameNet, PropBank and KYOTO
428 Annotations: rdfs:label "role "@en,
429              rdfs:comment "Open enumeration of roles, taken from FrameNet, PropBank or KYOTO."@en
430
431 ObjectProperty: nwr:dep
432 Domain: nwr:Participation Range: nwr:Dependency Characteristics: Functional
433 Annotations: rdfs:label "dependency"@en,
434              rdfs:comment "Specifies the kind of dependency among the object and event mentions in the scope of a
435                  participation mention."@en
436
437 Class: nwr:Dependency
438 EquivalentTo: { nwr:en_subj, nwr:en_obj, nwr:en_indcompl, nwr:en_predcompl_subj, nwr:en_predcompl_obj,
```

```
                nwr:en_rmod, nwr:en_subjpass, nwr:en_indcomplpass, nwr:en_undef, nwr:es_subj , nwr:es_obj ,
                nwr:es_indcompl, nwr:es_predcompl_subj , nwr:es_predcompl_obj , nwr:es_rmod, nwr:es_subjpass ,
                nwr:es_indcomplpass, nwr:es_undef , nwr:du_subj, nwr:du_obj, nwr:du_indcompl,
                nwr:du_predcompl_subj, nwr:du_predcompl_obj, nwr:du_rmod, nwr:du_rmod, nwr:du_subjpass,
                nwr:du_indcomplpass, nwr:du_undef, nwr:it_subj , nwr:it_obj , nwr:it_indcompl ,
                nwr:it_predcompl_subj , nwr:it_predcompl_obj , nwr:it_rmod, nwr:it_subjpass , nwr:it_indcomplpass ,
                nwr:it_undef }
Annotations: rdfs:label "dependency"@en,
             rdfs:comment "Enumeration of dependencies, specific to each language."@en

Individual: nwr:en_subj          Individual: nwr:en_obj          Individual: nwr:en_indcompl
Individual: nwr:en_predcompl_subj Individual: nwr:en_predcompl_obj Individual: nwr:en_rmod
Individual: nwr:en_subjpass      Individual: nwr:en_indcomplpass Individual: nwr:en_undef
Individual: nwr:es_subj          Individual: nwr:es_obj          Individual: nwr:es_indcompl
Individual: nwr:es_predcompl_subj Individual: nwr:es_predcompl_obj Individual: nwr:es_rmod
Individual: nwr:es_subjpass      Individual: nwr:es_indcomplpass Individual: nwr:es_undef
Individual: nwr:du_subj          Individual: nwr:du_obj          Individual: nwr:du_indcompl
Individual: nwr:du_predcompl_subj Individual: nwr:du_predcompl_obj Individual: nwr:du_rmod
Individual: nwr:du_rmod          Individual: nwr:du_subjpass     Individual: nwr:du_indcomplpass
Individual: nwr:du_undef         Individual: nwr:it_subj         Individual: nwr:it_obj
Individual: nwr:it_indcompl      Individual: nwr:it_predcompl_subj Individual: nwr:it_predcompl_obj
Individual: nwr:it_rmod          Individual: nwr:it_subjpass     Individual: nwr:it_indcomplpass
Individual: nwr:it_undef

# STATEMENTS

Class: ks:Statement
Annotations: ks:storedAttribute nwr:crystallized ,
             ks:storedAttribute nwr:confidence,
             ks:storedAttribute <http://purl.org/dc/terms/source>,
             ks:storedAttribute <http://www.w3.org/2000/01/rdf-schema#comment>

DataProperty: nwr:crystallized
Domain: ks:Statement Range: xsd:boolean Characteristics: Functional
Annotations: rdfs:label "crystallized "@en,
             rdfs:comment "Specifies whether a statement has been crystallized (i.e., it can be considered as
                 background knowledge)."@en

DataProperty: nwr:confidence
Range: xsd:decimal
Annotations: rdfs:label "confidence"@en,
             rdfs:comment "Specifies a confidence value on a 0-1 scale."@en

# CONTEXTS

Class: ks:Context
Annotations: ks:storedAttribute sem:accordingTo,
             ks:storedAttribute sem:hasBeginTimeStamp,
             ks:storedAttribute sem:hasEndTimeStamp

# DISJOINTNESS CONSTRAINTS

DisjointClasses: ks:Resource, ks:Mention, ks:Entity , ks:Statement, ks:Context, nfo:FileDataObject , nwr:NAFLayer,
    nwr:ValueType, nwr:SyntacticType, nwr:ReferenceType, nwr:EntityType, nwr:EventType, nwr:PartOfSpeech,
    nwr:Tense, nwr:Aspect, nwr:VerbForm, nwr:Polarity , nwr:Mood, nwr:TIMEX3Type, nwr:TLinkType, nwr:Dependency
DisjointClasses: nwr:News, nwr:NAFAnnotation
DisjointClasses: nwr:SignalMention, nwr:ValueMention, nwr:EntityMention, nwr:RelationMention
DisjointClasses: nwr:ObjectMention, nwr:TimeOrEventMention
DisjointClasses: nwr:EventMention, nwr:TimeMention
DisjointClasses: nwr:TLink, nwr:CLink, nwr:SLink, nwr:Participation
```