The KnowledgeStore: A Storage Framework for Interlinking Unstructured and Structured Knowledge
Francesco Corcoglioniti, FBK-irst, Trento, Italy
Marco Rospocher, FBK-irst, Trento, Italy
Roldano Cattoni, FBK-irst, Trento, Italy
Bernardo Magnini, FBK-irst, Trento, Italy
Luciano Serafini, FBK-irst, Trento, Italy

The Graph Signature: A Scalable Query Optimization Index for RDF Graph Databases Using Bisimulation and Trace Equivalence Summarization
Mustafa Jarrar, Sina Institute, Birzeit University, Birzeit, Palestine
Anton Deik, Sina Institute, Birzeit University, Birzeit, Palestine

Towards Linked Open Business Registers: The Application of the Registered Organization Vocabulary in Greece
Anastasia Varytimou, International Hellenic University, Kentriki Makedonia, Greece
Nikolaos Loutas, PwC, Zaventem, Belgium
Vassilios Peristeras, EC, DG for Informatics, Interoperability Solutions for European Public Administrations, Brussels, Belgium
ABSTRACT

Although the quantity of structured information on the Web and within organizations is increasing, the majority of information remains available only in unstructured form. While different in form, both unstructured and structured information sources provide information about entities in the world and their properties and relations; still, frameworks for their seamless integration have not been deeply investigated. In this paper the authors describe the KnowledgeStore, a scalable, fault-tolerant, and Semantic Web grounded open-source storage system for interlinking structured and unstructured data. They present the concept, design, function and implementation of the system, and report on its concrete usage in three application scenarios within the NewsReader EU project, where it stores and supports the querying of millions of news articles interlinked with millions of RDF triples extracted from text and imported from Linked Open Data sources. The authors report on data population and data retrieval performances of the system measured through a number of experiments, and they also discuss the practical issues and lessons learned from these experiences.

Keywords: KnowledgeStore, Linked Open Data Sources, NewsReader EU Project, Structured Information, Unstructured Information

1. INTRODUCTION

With Semantic Web (SW) technologies coming of age and the public acclaim of the Linked Open Data (LOD) initiative, the last few years have seen a massive proliferation of structured data, both on the Web and within organizations. Nonetheless, the majority of information remains available only in unstructured form. While different in form, both unstructured and structured information sources provide information about entities in 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, and content available in one form may help in better interpreting the information.
contained in the other, something that may turn out to be crucial in applications where having "complete" knowledge is a requirement (e.g., situations where users have to make potentially critical decisions).

The last decades achievements in Natural Language Processing (NLP) now enable the large scale extraction of knowledge about world entities from unstructured text (Weikum & Theobald, 2010; Grishman, 2010), thus setting the basis to combine knowledge coming both from unstructured and structured content. However, the development of frameworks enabling the seamless integration and linking of knowledge available in structured and unstructured forms has only been partially investigated.

In this paper we present the KnowledgeStore, a scalable, fault-tolerant, and Semantic Web grounded storage system to jointly store, manage, retrieve, and query both structured and unstructured data. To illustrate the capabilities and peculiarities of the KnowledgeStore, let us consider the following scenario. Among a collection of news articles, a user is interested in retrieving all 2014 news reporting statements of a 20th century US president where he is positively mentioned as "commander-in-chief". On one side, the KnowledgeStore supports storing of resources (e.g., news articles) and their relevant metadata (e.g., the publishing date of a news article). On the other side, it enables storing structured content about entities of the world (e.g., the fact of being a US president, the event of making a statement), either extracted from text or available in LOD/RDF datasets (e.g., DBpedia, Yago), in a contextualized fashion (e.g., someone is US president only for a certain period of time). And last, through the notion of mention, it enables linking an entity or fact of the world to each of its occurrences in documents, allowing also to store additional information (mention attributes, typically extracted while processing the text) for each specific occurrence in a document: to name a few, the position of the entity/fact in the text (e.g., between character 1022 to 1040), the explicit way it occurs (e.g., "commander-in-chief"), and the sentiment of the article writer on that particular occurrence (e.g., positively mentioned). Besides supporting the storage and management of this content, the KnowledgeStore provides query and retrieval mechanisms that enable to access all the information it contains and can be used to answer the user query presented above.

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 favors the design and empirical investigation of information processing tasks otherwise difficult to experiment with. On the one hand, the possibility to semantically query the content of the KnowledgeStore with requests combining knowledge from structured sources and unstructured sources, similarly to the example previously discussed, allows a deeper exploration and analysis of stored data, a capability particularly useful in applications such as decision support. On the other hand, the joint storage of structured knowledge (both background and extracted knowledge), the resources it derives from, and mention information — all effectively accessible through a single API — provides an ideal scenario for developing, debugging, training, and evaluating tools for a number of NLP and knowledge processing tasks. NLP tasks can benefit from the availability of background knowledge and the textual grounding of mentions, exploiting them to improve their performance: an example is coreference resolution (i.e., identifying that two mentions refer to the same entity of the world), especially in cross-document settings. Similarly, knowledge processors can exploit the linking of structured knowledge to mentions, and the linguistic features attached to them, to perform tasks such as knowledge fusion (i.e., the merging of possibly contradicting information extracted from different sources).

Though a few proposals have been made for frameworks enabling the seamless integration and linking of knowledge coming both from structured and unstructured contents (Popov, Kiryakov, Kirilov, Manov, & Goranov, 2003; Gönül & Sinaci, 2012; Kurz, Güntner, Damjanovic, Schaf-
fert, & Fernandez, 2014), the KnowledgeStore is novel in several aspects: (i) differently from other approaches where a mere link between entities and documents is maintained, it explicitly accounts for mentions and mention attributes, thus allowing to exploit them in applications and enhanced processing tasks; (ii) by explicitly storing the link entities/facts—mentions—resources, it favors a fine-grained tracking of the provenance of entities and facts to the sources stating them; and (iii) it enables to contextualize facts, thus making explicit the circumstances (e.g., time, point of view) in which those facts hold.

The idea behind the KnowledgeStore was preliminarily investigated in Cattoni et al. (2012) and tested in the scope of the LiveMemories project. However, in Corcoglioniti, Rospocher, Cattoni, Magnini, and Serafini, (2013) we greatly revised the initial design, introducing significant enhancements: (i) support for storing events and related information, such as event participants (the previous version was limited to mentions and entities referring to persons, organizations, geo-political entities, and locations) according to Semantic Web best practices, (ii) a new architecture that favors scaling on a significantly larger collection of resources (potentially, millions of documents versus a few hundreds of thousands), (iii) a semantic querying mechanism over its content, and (iv) a HTTP ReST API as well as a web user interface to seamlessly inspect its content. Besides some refinement of the KnowledgeStore design, in this paper we significantly extend the work in Corcoglioniti et al. (2013), presenting a first complete, used in practice, and evaluated implementation of the KnowledgeStore. In details, the contributions of this paper include:

- The revised version of the KnowledgeStore conceptual model and architectural design (w.r.t. Corcoglioniti et al., 2013) in view of the experience gained in implementing it;
- The description of the first complete implemented version of the KnowledgeStore, released open-source under the terms of the Apache License v2.0;
- The description of a number of concrete usage scenarios in the scope of the NewsReader EU Project, where the KnowledgeStore was populated with various large-size datasets and a number of applications (mainly for decision support) were successfully built on top of it, thus demonstrating the capabilities of the KnowledgeStore;
- A thorough evaluation of the performances of the KnowledgeStore, covering both data population and data retrieval with different dataset sizes and numbers of concurrent clients;
- A discussion of lessons learned and open issues based on our implementation and usage experience, applicable to the KnowledgeStore but also to any system addressing the same goals.

The paper is organized as follows. Section 2 provides an overview of the conceptual model underlying the KnowledgeStore as well as its architectural role in systems dealing with structured and unstructured contents. Section 3 describes the KnowledgeStore system, focusing on its data model, architecture, API and user interface, while Section 4 discusses its concrete use in the NewsReader Project, showing how multiple KnowledgeStore instances have been configured, populated and their contents accessed in three application scenarios. Section 5 provides a more in-depth evaluation of the data population and data retrieval performances of the system, presenting the results of a number of experiments. Based on these experiments and the NewsReader experience, Section 6 summarizes relevant lessons learned and open issues. Section 7 briefly overviews related state-of-the-art approaches, while Section 8 concludes with some final remarks.
2. THE KNOWLEDGESTORE: CONCEPTUAL OVERVIEW

To support the storage and alignment of knowledge of unstructured and structured information sources, the KnowledgeStore internally adopts a three-layer content organization, as schematically shown in Figure 1.

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), each having a representation (e.g., a text file) and some descriptive metadata (e.g., title, actor, document creation date). On the other hand, the entity layer is the home of structured content, which, based on Knowledge Representation and Semantic Web best practices, consists of axioms\(^6\) (sets of \(<\text{subject}, \text{predicate}, \text{object}>\) triples) describing the entities of the world (e.g., persons, locations, events), and for which additional metadata are kept to track their provenance and to denote the formal contexts where they hold (e.g., time validity, point of view, attribution). Between the aforementioned two layers there is the mention layer, which indexes mentions, i.e., snippets of resources (e.g., fragments of text) that denote something of interest, such as an entity or an axiom of the entity layer. Clearly, a resource may contain many mentions, each one associated to some entity or axiom, and an entity or axiom may have multiple associated mentions within the same resource, denoting the fact that the entity or axiom occurs multiple times in it.

The explicit representation of mentions has a conceptual grounding in the need for formally distinguishing entities (aka, referents, reference) from their actual representations (aka, signs, senses), a topic extensively debated in philosophy of language.\(^7\) Furthermore, differently from the other state-of-the-art approaches (cf. Section 7) that typically only highlight the key roles

---

Figure 1. The three representation layers in the KnowledgeStore: Resource, Mention, and Entity

---

"Stripes," Dole, 1996

He (Bill Clinton) quickly became the first civilian commander-in-chief to salute his marine guards while entering or exiting an aircraft.

- dbpedia:Bill_Clinton
- rdf:type: yago:Politician110451263
- rdfs:label: "Bill Clinton"@en
- dbo:birthPlace: dbpedia:Hope,_Arkansas
of documents and entities, and treat implicitly mentions as plain links between them (2-layer approaches), mentions can be extensively described in the KnowledgeStore by means of attributes. These attributes allow storing of relevant information that is specific to the mention, instead of the enclosing resource or the entity it denotes; examples are the extent with which the entity/axiom is referred in the resource, or a sentiment value extracted from the mention extent that expresses the attitude of the article writer towards the denoted entity/fact. This representation choice has indeed some theoretical roots: many philosophical accounts (e.g., Frege, 2000) treat signs as rich objects having not just a referent (the ‘bedeutung’, in Frege words) but also a sign-specific sense (the ‘sinn’) characterizing the way the sign denotes its referent. For instance, mentions such as “Barack Obama”, “The president of the USA” and “Nobel peace prize winner” have the same referent (the latter two under certain circumstances) but different senses (as of Frege) as they highlight different roles and aspects of the referent. As this kind of information, typically extracted by NLP tools while processing the resource, is mention-specific (for a given mention of that entity/axiom, in that particular document), we preferred to represent and store mentions (and their attributes) in a principled way, elevating them as first class objects in the KnowledgeStore.

From an architectural point of view, the KnowledgeStore can be seen as a centralized service accessed by external clients for storing and retrieving the content they process and produce. These clients can be classified in three types, according to their main function (see Figure 2):

- **Populators**: Their purpose is to feed the KnowledgeStore with basic content. We distinguish two main typologies of populators: *unstructured populators* that pre-load in the KnowledgeStore resources like textual documents, web-pages, and so on; and, *structured populators* that inject in the KnowledgeStore structured content obtained from LOD datasets or, more generally, RDF repositories.
- **Information Extraction Processors**: They process resources uploaded by the populators to extract mentions and entities from them. A number of NLP tools and suites can be readily used for this purpose, like the ones used in the NewsReader project and publicly available.

Figure 2. Interactions with external modules
online (see Section 4). Information extraction processors can be classified as either single-resource or cross-resource. **Single-resource processors** perform tasks defined at the level of a resource or of a portion of it (e.g., a sentence), such as part-of-speech tagging, dependency parsing and semantic role labeling; for these tasks, the processing of a resource is independent of the processing of other resources and thus multiple resources can be processed in parallel on a number of machines. **Cross-resource processors**, on the other hand, perform tasks defined on whole collections of resources, such as cross-document coreference resolution or information clustering/summarization; these tasks typically combine information from multiple resources, cannot be easily parallelized and their cost increases with the dataset size. Both kinds of processors may actually exploit the KnowledgeStore as a data-hub, using it to exchange data between them.

- **Applications**: Like decision support systems and enhanced web-based applications, that mainly read data from the KnowledgeStore offering services on top of its content.

Note that the KnowledgeStore does not enforce a particular client interaction paradigm for what concerns the content access and population. In particular, content can be (i) injected in one shot, and subsequently accessed by applications in a sort of “read-only” mode (write once, read many), or (ii) continually incrementally added (as in case of a daily feed of news articles), where clients work more in a sort of “stream-oriented” processing mode (write many, read many).

## 3. SYSTEM DESCRIPTION

Next we introduce the main features and components of the KnowledgeStore. Additional documentation, a demo video showcasing a navigation through the KnowledgeStore content, as well as binaries and source code of the KnowledgeStore, are available on the KnowledgeStore website. A running KnowledgeStore instance is also publicly accessible.

### 3.1. The KnowledgeStore Data Model

The KnowledgeStore data model defines what information can be stored in the KnowledgeStore. As pointed out in Section 2, it is centered on the resource, mention and entity layers. Resources and mentions are described using a configurable set of types, attributes and relations. Entities are described with an open set of axioms enriched with metadata attributes (e.g., for context and provenance). The KnowledgeStore data model is formalized as an OWL 2 (Motik, Parsia, & Patel-Schneider, 2009) ontology accessible online, with types, attributes, and relations identified via URIs; terms from the Nepomuk Information Element vocabulary (prefix nie:), the Nepomuk File Ontology (prefix nfo:), and the Grounded Annotation Framework ontology (gaf:*), are reused in the ontology. The UML class diagram of Figure 3 summarizes the main aspects of the KnowledgeStore data model.

Flexibility is a key requirement for the data model, as (i) different kinds of unstructured and structured content can be stored in different KnowledgeStore instances; and (ii) the kind of information stored in a KnowledgeStore instance may evolve in time. For this reason, the data model is divided in a **fixed part**, embodied in the implementation and kept as small as possible, and a **configurable part** that is specific to each KnowledgeStore instance and is used to organize and fine tune its storage layout. More in details, the fixed part includes:

- The Resource, Mention and Entity classes;
• The Axiom class with its RDF encoding (encodedBy property) and the Context it holdsIn;
• The Representation of a resource, including its file and metadata managed by the system;
• The relation storedAs, linking a resource to its representation;
• The relation hasMention, linking a resource to the mentions it contains;
• The relation gaf:denotedBy (Fokkens et al., 2014), linking an entity or axiom to the mention(s) expressing it, used to track provenance\(^{17}\) and to debug information extraction pipelines used with the KnowledgeStore.

For a given specific application, the KnowledgeStore data model can be manually customized by defining another OWL 2 ontology, specific to that KnowledgeStore instance, which imports and extends the configurable part of the data model by refining the TBox definitions of the KnowledgeStore ontology. Typically, this consists in defining:

• The subclass hierarchy of Resource and Mention;\(^{18}\)
• Additional attributes of Resource, Mention, Axiom and their subclasses; additional relations among resources or among mentions;
• Enumerations and classes used as attribute types (similarly to Representation); restrictions on fixed part relations (not shown in figure).

This modular approach enables to accommodate very different configurations: from KnowledgeStore instances where mentions are just pointers for entities to the characters in the resources where they are referred (in this special case, the KnowledgeStore basically downgrades to a standard 2-layer resources/entities framework), to more enhanced instances where a very rich set of linguistic attributes is stored for each mention. A concrete example of application-specific customization of the KnowledgeStore data model is presented in Section 4.

It is worth noting that the choice of rooting the data model in OWL 2 and using an OWL 2 ontology for its configuration provides a number of benefits. First, it allows both the model definition and the instance data to be encoded in RDF (Beckett, 2004), enabling the use of Semantic Web technologies for manipulating them and their publication on the Web according to the Linked Open Data best practices. Second, to some extent, data validation can be performed using an OWL 2 reasoner. In this case, it must be noted that resource and mention instances form a huge ABox. Some rule-based reasoners, such as RDFox (Motik, Nenov, Piro, Horrocks, & Olteanu, 2014), support OWL 2 RL reasoning over large ABoxes, but their memory requirements would pose a limit on the scalability of the system.\(^{19}\) This problem can be tackled
by performing reasoning on a per-resource (and its mentions) basis, exploiting the fact that resource descriptions are largely independent one to another. Of course, this solution sacrifices completeness of reasoning for scalability, but at the same it enables the use of OWL 2 profiles more expressive than OWL 2 RL, and even of OWL 2 extensions (Patel-Schneider & Franconi, 2012; Tao, Sirin, Bao, & McGuinness, 2010) that realize a restricted closed world assumption useful for validation purposes.

3.2. The KnowledgeStore API

The KnowledgeStore presents a number of interfaces, offered as part of the KnowledgeStore API, through which external clients may access and manipulate stored data. These interfaces are available through two HTTP ReST endpoints.

The CRUD endpoint provides the basic operations to access and manipulate any object stored in any of the layers of the KnowledgeStore; for instance, Figure 4 shows the HTTP invocation of a retrieve operation returning all the resources with dct:publisher being equal to dbpedia:TechCrunch.

Several aspects (e.g., operation granularity, transactional properties, access control) have been considered in defining the operations provided by the CRUD endpoint. For efficiency reasons, the KnowledgeStore offers coarse-grained streaming operations that operate on multiple objects at once (e.g., the simultaneous update of all the mentions of a certain resource). As having fully transactional operations is unfeasible (as an operation can potentially affect all the KnowledgeStore content) and perhaps unwanted (e.g., on an update operation on 1 million objects, failing on a particular object should not cause the rollback of the operation for the other objects), a coarse-grained API call behaves in a transactional way and satisfies ACID properties only on each single object handled in the call (e.g., a single element in a set of mentions).

The SPARQL endpoint allows querying of axioms in the entity layer using the SPARQL query language, a W3C standard for retrieving and manipulating data in Semantic Web repositories. This endpoint provides a flexible and Semantic Web-compliant way to query for entity data, and leverages the grounding of the KnowledgeStore data model in Knowledge Representation and Semantic Web best practices.

For both endpoints, access control is employed to restrict usage of the KnowledgeStore API and content only to authorized clients. Authentication is based on separate username/password credentials.

Figure 4. Invocation of CRUD retrieve operation through the HTTP ReST endpoint

curl -request GET http://newsreader.fbk.eu/ktest/resources.rdf?\&\r\n\n   dcterms:isReferencedBy = dbpedia:TechCrunch (*)

(* URL encoding omitted)

```xml
<rdf:RDF 
   xmlns:nwr="http://dkm.fbk.eu/ontologies/newsreader#" ...>
   <nwr:News rdf:about="http://newsreader.fbk.eu/resources.rdf/r105"> 
     <dcterms:issued>2013-09-30</dcterms:issued>
     <nfo:fileName>r105.txt</nfo:fileName>
     <nfo:fileSize>15012</nfo:fileSize>
     <nfo:fileCreated>2013-09-30</nfo:fileCreated>
     <nfo:mimeType>text/plain</nfo:mimeType>
   </nwr:News>
   ... 
</rdf:RDF>
```
credentials for each authorized client, while access may be limited to restricted parts of the KnowledgeStore content (e.g., only the mention layer, only resources from a certain provider). While the two HTTP ReST are language- and platform-neutral and thus allow the integration of the KnowledgeStore in any computing environment, for clients developed in Java a specific client library is also offered to ease the interaction with the KnowledgeStore and take care of the optimal use of the two endpoints.

### 3.3. The KnowledgeStore Architecture

The internal KnowledgeStore architecture is centered around the KnowledgeStore Server (Figure 5), a specifically developed software component that implements the operations of the CRUD and SPARQL endpoints, handling global issues such as access control, data validation and operation transactionality; it also provides the KnowledgeStore UI.

Data storage is delegated by the KnowledgeStore Server to three software components distributed on a cluster of machines: Hadoop HDFS, HBase and Virtuoso. The Hadoop HDFS\(^\text{23}\) distributed filesystem provides a reliable and scalable storage for the files holding the representations of resources (e.g., texts and linguistic annotations of news articles). HDFS provides transparent data distribution and replication and fault tolerance with respect to single node failures. The HBase\(^\text{24}\) column-oriented store provides database services for storing and retrieving semi-structured information about resources and mentions. HBase builds on Hadoop HDFS and inherits its reliability and scalability characteristics, being particularly suited for random, real time read/write access to huge quantities of data, when the nature of data does not require a relational model (like in the case of resource and mention data). In the current setup, each resource and mention is stored in HBase as a row indexed by its URI. This solution allows for optimal lookup performances, and is insensitive to the number of mentions per resource, i.e., it works equally well with very small and very large resources—what matters it the total number of objects stored.\(^\text{25}\) On the other hand, retrieval by filter condition on one or more attributes often requires full table scans, a situation that is mitigated by the possibility to distribute and parallelize such scans over all the nodes forming the HBase cluster.

The Virtuoso\(^\text{26}\) triplestore indexes axioms’ triples to provide services supporting reasoning and online SPARQL query answering. Virtuoso has been selected motivated by its excellent performance in recent benchmarks (e.g., the April 2013 BSBM benchmark\(^\text{27}\)), further improved

![Figure 5. KnowledgeStore Components. Hadoop and HBase comprise multiple, distributed processes, while the KnowledgeStore Server and Virtuoso are single processes](image-url)
in the latest releases, as well as for its availability for both a single-machine and a cluster deployment configuration. Axioms are stored in Virtuoso as sets of <subject, predicate, object> RDF triples within named graphs that define the context of validity of these triples and allow tracking of their provenance. Virtuoso supports a limited form of query-time RDFS reasoning but we do not use it. Instead, we perform forward-chaining RDFS reasoning to materialize inferable axioms, keeping track of their provenance using named graphs. For the time being, we limit reasoning to background knowledge only (i.e., axioms coming from LOD resources), as we are still working towards a principled and scalable solution for dealing with the noise and inconsistencies in entity data coming from NLP processing. To that respect, we remark that our data organization based on the use of named graphs is consistent with the Contextualized Knowledge Repository (CKR) representation model (Bozzato & Serafini, 2013), and indeed we are evaluating the deployment of this reasoning paradigm (or an extension of it) on top of the KnowledgeStore to properly combine information with different contextual and quality characteristics.

Concerning the choice of the storage backend(s), it is worth noting that none of Hadoop HDFS, HBase and Virtuoso can store all the data alone, and hence their combination is crucial to realize a hybrid storage system like the KnowledgeStore. In fact, the use of a triplestore for entity axioms currently represents the state-of-the-art choice for providing efficient SPARQL access to this kind of data. At the same time, mention and resource data are too voluminous to be stored in the triplestore and needs an additional storage backend (HBase). Finally, large textual content is poorly supported in triplestores and databases (both relational and NoSQL), and is best stored in a filesystem or similar structure (Hadoop HDFS).

Finally, not shown in Figure 5 are additional tools and scripts to: (i) enforce the transactional guarantees of the KnowledgeStore API operations; (ii) synchronize access and management of HBase nodes; and (iii) manage the complexity of software deployment in a cluster environment (e.g., the management scripts for infrastructure deployment, start-up & shut-down, data backup & restoration, and gathering of statistics).

3.4. The KnowledgeStore User Interface

While the KnowledgeStore can be programmatically accessed by clients through its API, human users can easily interact with the KnowledgeStore through the KnowledgeStore User Interface (UI). The KnowledgeStore UI is a basic web-based application whose main purpose is to enable users to inspect and navigate the KnowledgeStore content without having to develop applications accessing the KnowledgeStore API. Two core operations are offered:

- The **SPARQL query** operation, with which arbitrary SPARQL queries can be run against the KnowledgeStore SPARQL endpoint, obtaining the results directly in the browser or as a downloadable file in various formats. Figure 6c shows an excerpt of the results obtained by running a query in the SPARQL tab of the KnowledgeStore UI;
- The **lookup operation**, which, given the URI of an object (i.e., resource, mention, entity), retrieves all the KnowledgeStore content about that object. Figure 6a and Figure 6b show the output obtained by running a lookup operation for a resource and for a mention.

These two operations are seamlessly integrated in the UI, to offer a smooth browsing experience to the users. For instance, it is possible to directly invoke the lookup operation on any entity returned in the result set of a SPARQL query. Similarly, when performing the lookup operation on a resource, all mentions occurring in the resource are highlighted (see the “Resource text” box in Figure 6a) with a different color for the various mention types (e.g., person, organization,
Figure 6. The KnowledgeStore UI

(a) Resource Lookup

(b) Mention Lookup

(c) Running a SPARQL query

location, event), and by clicking on any of them the user can access all the details for that mention (see Figure 6b). Finally, the lookup of a mention (see Figure 6b) returns the attributes of the selected mention (box Mention data) as well as its occurrence in the containing resource (box Mention resource) and the structured description of the real-world entity it refers to (box Ment-
tion referent), capturing in a single page the three representation layers of the KnowledgeStore as well as the role of mentions as a bridge between unstructured and structured content.

4. THE KNOWLEDGESTORE IN ACTION

In this section we report the successful deployment of several KnowledgeStore instances populated with real content, extracted from news corpora varying in size (from 18K to 1.3M news articles) and covering different domains. Some of these instances, deployed in the context of the NewsReader EU project, were exploited to build enhanced applications to support decision making and data journalism.

4.1. The NewsReader EU Project

The goal of the NewsReader project (Jan 2013 - Dec 2015) is to process news (e.g., 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 users (e.g., professional decision-makers), that by means of visual interfaces and interaction mechanisms are able to explore them, exploiting their explanatory power and their systematic structural implications, to make well-informed decisions. Achieving this challenging goal requires NewsReader to address several specific objectives:

- 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 corefer mentions of the same entity;
- To complete entity descriptions by complementing extracted mention information with available structured background knowledge (e.g., DBpedia, corporate databases);
- To interrelate entities (events and their participants, in particular) to support the construction of narrative stories;
- To reason over events (logically and/or statistically) 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.

4.2. The KnowledgeStore in NewsReader

The KnowledgeStore plays a central role in addressing the objectives of the NewsReader project: the NewsReader information extraction pipelines process news articles populating the KnowledgeStore with structured knowledge, while applications built on top of the KnowledgeStore present the users with comprehensive views on the heterogeneous content stored in it. More in details, the KnowledgeStore has been successfully deployed, populated, and exploited to build enhanced applications in a number of concrete NewsReader scenarios, in different domains, with real content varying in size. Before detailing these scenarios, we describe the KnowledgeStore data model configuration adopted in NewsReader, as well as the processing tools used to populate the various KnowledgeStore instances.
Figure 7 shows how the KnowledgeStore data model was manually configured for the NewsReader scenarios.

The original news articles, together with their corresponding annotated versions obtained by processing them with NLP information extraction tools, are stored in the resource layer, each described by metadata from the Dublin Core and Nepomuk vocabularies. Several types of mentions are stored, which denote either an entity (e.g., person, organization, event), a relation among entities (e.g., participation links between event and participant mentions, as well as causal, temporal and subordinate links among event mentions and / or time expressions, derived from the TimeML standard defined in Pustejovsky, Lee, Bunt, and Romary, 2010), or a numerical quantity. The NLP Interchange Format (NIF) vocabulary has been used to define basic mention properties, thus enabling interoperability with tools consuming NIF data; in addition, several specific attributes have been added to store information extracted from NLP processing. At the entity level, provenance, confidence, and contextual metadata have been associated to statements, exploiting also the Simple Event Model (SEM) vocabulary (van Hage, Malaisé, Segers, Hollink, & Schreiber, 2011). The OWL ontology of the NewsReader KnowledgeStore data model is available for download.

The KnowledgeStore population in NewsReader is organized in two main phases, in accordance with the two information processors developed in the project: a single-resource processor, the NewsReader NLP pipeline, and a cross-resource processor, the VUA Event Coreference Module.

The NewsReader NLP pipeline processes each news article provided in input, enriching it with NAF (NLP Annotation Format, Fokkens et al., 2014) annotations about: tokenization, lemmatization, part-of-speech tagging, parsing, word sense disambiguation, named entity linking to DBpedia, semantic role labelling, nominal coreference, temporal expression recognition, opinion mining, and event coreference. At the end of the NewsReader NLP pipeline, the KnowledgeStore

Figure 7. KnowledgeStore data model configured for the NewsReader scenarios
NAF populator is invoked to upload in the KnowledgeStore resource layer the complete NAF annotated version of the source news article, and to inject in the KnowledgeStore the mentions (and their metadata) extracted by processing the news article. The NAF populator is also used to upload into the KnowledgeStore resource layer all the source news articles, setting the values of the several metadata attributes attached to each news article (e.g., publication date, author, title).

The VUA Event Coreference Module works on the results produced by the NewsReader NLP pipeline by processing all the mentions extracted. Clusters of mentions referring to the same entity (e.g., event, person, organization) are identified using machine learning techniques and a number of features, including mention extents, links to DBpedia, and the event-actor links from semantic role labeling (more details in Cybulska and Vossen, 2014). An entity is created for each cluster of mentions and axioms describing and linking these entities are asserted based on attributes and relations in the mention layer. These entities and axioms are injected into the KnowledgeStore via RDFpro (Corcoglioniti, Rospocher, Mostarda, & Amadori, 2015), a tool for RDF stream processing. RDFpro is also used to populate the KnowledgeStore with background knowledge, i.e., RDF content directly injected into the KnowledgeStore entity layer, that may (i) support some tasks performed by the information extraction processors, and (ii) complement the information automatically extracted from news with quality content available in structured resources such as DBpedia, Freebase, and GeoNames, to favor the exploitation of the KnowledgeStore content by applications built on top of it.

Next we present the different scenarios where the populated KnowledgeStore instances were deployed.

Table 1 summarizes the characteristics of the various KnowledgeStore instances, in particular in terms of size (number of objects in the three KnowledgeStore layers with a more detailed breakdown for entities and axioms, and disk space occupation) and population time.

Details on the domain, news article providers, and period when each instance was prepared are also reported.

To put these numbers in perspective, we remark that differences across the instances, especially in interpreting extracted information or population times, could be also due to different versions of information extraction processors, populators, and KnowledgeStore software used: the latest stable versions of these tools were used at the time of populating each KnowledgeStore instance. Also, note that different versions of background knowledge were injected in the various KnowledgeStore instances, to accommodate specific needs of the scenario considered or to use the latest official release of the original dataset.

### 4.3. Scenario 1: Wikinews

This first scenario is about supporting benchmarking of the NewsReader information extraction processors.

A KnowledgeStore instance (cf. column “Wikinews” in Table 1) was populated with 18,510 general domain news from Wikinews (news time period: 2004-2014).

Given its controlled size, substantially smaller than the other KnowledgeStore populations here reported, this instance is used project-wise to benchmark and improve the performances of the NewsReader information extraction processors, by comparing their outputs with gold standard annotations produced by a team of linguists as part of the project.

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

4.4. Scenario 2: FIFA 2014 World Cup

The second scenario is about revealing hidden facts and people networks behind the FIFA World Cup (2014), by building web-based applications on top of the KnowledgeStore.

A total of 212,258 football-related news articles, from various providers (including BBC and The Guardian) and distributed over a time period of ten years (2005-2014), were processed and uploaded into the KnowledgeStore (cf. column “FIFA World Cup” in Table 1). While data collection and preparation required significant time and effort, the development of applications on top of stored content was realized as part of a Hack Day event,43 where 40 people, a mixture of Linked Data enthusiasts and data journalists, gathered for one day to collaboratively develop web-based applications on top of the KnowledgeStore. Ten web-based applications, implemented in different programming languages, were developed in roughly 6 working hours. The development was facilitated, especially for people not familiar with Semantic Web technologies such as RDF and SPARQL, by the availability of the NewsReader Simple API44 (Hopkinson, Maude, & Rospocher, 2014), an HTTP ReST API developed in python that uses JSON and is easily accessible from JavaScript, and where each method implements a SPARQL query template instantiated at runtime with the actual parameters passed to the method (e.g., the method “Actors of a specified type” implements a query that returns all instances having as RDF type the value of the parameter passed to the method). Each application was developed with a focused purpose: among them, to determine which teams some football player had played during his career (by looking at transfer events); to discover which football teams were most commonly associated

<table>
<thead>
<tr>
<th>Domain</th>
<th>WikiNews</th>
<th>Cars (Ver. 1)</th>
<th>FIFA WorldCup</th>
<th>Cars (Ver. 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>News Providers</td>
<td>General News</td>
<td>Automotive Industry</td>
<td>Sport, Football</td>
<td>Automotive Industry</td>
</tr>
<tr>
<td>en.wikinews.org</td>
<td>LexisNexis</td>
<td>LexisNexis, BBC,</td>
<td>LexisNexis</td>
<td></td>
</tr>
<tr>
<td>Resources</td>
<td>37,020</td>
<td>127,270</td>
<td>424,516</td>
<td>2,519,496</td>
</tr>
<tr>
<td>NAF Documents</td>
<td>18,510</td>
<td>63,635</td>
<td>212,258</td>
<td>1,259,748</td>
</tr>
<tr>
<td>News Articles</td>
<td>18,510</td>
<td>63,635</td>
<td>212,258</td>
<td>1,259,748</td>
</tr>
<tr>
<td>Mentions</td>
<td>314</td>
<td>531</td>
<td>597</td>
<td>887</td>
</tr>
<tr>
<td>per News Article</td>
<td>2,629,176</td>
<td>9,110,683</td>
<td>76,165,114</td>
<td>205,114,711</td>
</tr>
<tr>
<td>Entities</td>
<td>951,879</td>
<td>2,212,691</td>
<td>10,246,338</td>
<td>27,123,724</td>
</tr>
<tr>
<td>Events</td>
<td>624,439</td>
<td>1,783,991</td>
<td>9,387,356</td>
<td>25,156,574</td>
</tr>
<tr>
<td>Persons</td>
<td>94,731</td>
<td>199,999</td>
<td>403,031</td>
<td>729,797</td>
</tr>
<tr>
<td>Organizations</td>
<td>101,754</td>
<td>187,842</td>
<td>431,232</td>
<td>947,262</td>
</tr>
<tr>
<td>Locations</td>
<td>130,955</td>
<td>40,859</td>
<td>24,729</td>
<td>290,091</td>
</tr>
<tr>
<td>Axioms (Triples)</td>
<td>105,675,519</td>
<td>316,034,616</td>
<td>240,731,408</td>
<td>535,035,576</td>
</tr>
<tr>
<td>from Mentions</td>
<td>9,706,585</td>
<td>46,359,300</td>
<td>136,125,841</td>
<td>439,080,042</td>
</tr>
<tr>
<td>from Background Knowledge distilled from</td>
<td>95,974,934</td>
<td>269,675,316</td>
<td>104,595,567</td>
<td>95,974,934</td>
</tr>
<tr>
<td>DBpedia 2014 (EN)</td>
<td>17.64</td>
<td>30.67</td>
<td>82.48</td>
<td>260.20</td>
</tr>
<tr>
<td>Resource Layer</td>
<td>1.25</td>
<td>3.10</td>
<td>16.55</td>
<td>108.27</td>
</tr>
<tr>
<td>Mention Layer</td>
<td>1.49</td>
<td>4.77</td>
<td>41.72</td>
<td>112.00</td>
</tr>
<tr>
<td>Entity Layer</td>
<td>14.90</td>
<td>22.80</td>
<td>24.21</td>
<td>39.93</td>
</tr>
<tr>
<td>Approx. Population Total Time (hrs)</td>
<td>2</td>
<td>30</td>
<td>56</td>
<td>160</td>
</tr>
<tr>
<td>Approx. Rate (news articles/h)</td>
<td>9,300</td>
<td>2,250</td>
<td>4,000</td>
<td>7,800</td>
</tr>
</tbody>
</table>

Table 1. Overview of the different KnowledgeStore instances (from smallest to largest, in terms of number of resources) exploited in the various scenarios; the Wikinews instance is publicly available.
with violence; to determine people and companies related to gambling; and, to establish the popularity of people, companies, and football teams in different locations.

During the Hack Day, the KnowledgeStore received 30,126 queries (on average, 1 query/second, with peaks of 20 queries/second), issued either directly through the SPARQL endpoint or via the NewsReader Simple API, and successfully served them on average in 654ms (only 40 queries out of 30,126 took more than 60 seconds to complete).

### 4.5. Scenario 3: Global Automotive Industry Crisis Review

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

Two KnowledgeStore instances were prepared, one in January 2014 (cf. “Cars (Ver. 1)” in Table 1-6,635 news articles), and one in December 2014 (cf. “Cars (Ver. 2)” in Table 1-1,259,748 news articles). In both cases, the news articles (time period: 2003-2013) were made available for project purposes by LexisNexis.45

The main application in this scenario is SynerScope,46 a visual analytics application that delivers real time interaction with dynamic network-centric data. SynerScope interacts with the KnowledgeStore through the KnowledgeStore exporter (van Hage & Ploeger, 2014), a tool that converts the data stored in the KnowledgeStore to the format digested by SynerScope. SynerScope offers different views (e.g., table view, hierarchical view, map view) on the KnowledgeStore content, enabling users to navigate it through various interaction methods (e.g., selection/highlight, drill down/up, expansion). This way, it is possible to visually browse all events that involve a given person or company, or to build networks of persons/companies based on event co-participation.

In addition to the SynerScope application, the capability to query the KnowledgeStore content was exploited to deliver automatically generated reports (and plots) supporting decision makers. For instance, by retrieving the different events involving the ten major car companies, it was possible to generate a report showing the trend of the quantity of events per year in which these companies were involved in the considered period, and therefore to assess their popularity (according to the considered dataset) during the economic crisis. Similarly, by retrieving the different events and locations and times they took place at, we were able to produce maps (one per year) to obtain insights into how the localities of the global automotive industry changed during the crisis.

The Cars (Ver. 2) KnowledgeStore instance has been also exploited in two Hack Day events,47 to build enhanced applications and conduct exploratory investigation on top of the KnowledgeStore: among them, an in-depth analysis of the age of CEOs when they get hired or fired from companies, analysis of most dangerous cars around, car companies with high cars recall rate, and so on.

During the two Hack Days, the KnowledgeStore received 118,094 requests (on average, 3 requests/second, with peaks of 40 requests/second), issued either directly through its endpoints or via the NewsReader Simple API, and successfully served them on average in 31ms.

### 5. PERFORMANCE ASSESSMENT

In this section we report the results of a number of experiments aimed at assessing the performances of the KnowledgeStore. We focus on two core operations that are relevant for the practical adoption of the system: data population, analyzed in Section 5.1, and data retrieval, analyzed in Section 5.2. All the experiments use real world data from the Cars (Ver. 2) dataset (Section 4.5)
and were conducted on a cluster of five servers connected by a Gigabit LAN: one running the KnowledgeStore Server and the Virtuoso triplestore and having 1 TB disk, 32 GB RAM and two Intel Xeon E5-2440 CPUs; the others running Hadoop and HBase and each having 1 TB disk, 32 GB RAM (except a server with 8 GB only) and two Intel Xeon E6545 CPUs.

5.1. Data Population Performances

As previously discussed, the KnowledgeStore offers a number of data upload (and manipulation) operations that support different population workflows. Here we consider the workflow adopted in NewsReader and described in Section 4.2., as we deem it representative of a broad range of usage scenarios.

In the considered workflow, both the resource and mention layers are populated first (e.g., using the NAF populator) with the results of the single-document information extraction processors; then, the entity layer is populated (e.g., using RDFpro) with the entities and axioms extracted by the cross-document information extraction processors, as well as with background knowledge. These two steps have very different performances: while the entity layer population is fast—400K axioms/s, corresponding to 4M news articles per hour given an average of 350 axioms per news article (Cars (Ver. 2) dataset)—the population of the resource and mention layers is around three order of magnitude slower—8K news articles per hour—and thus dominates and determines the overall population performances.

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

5.1.1. Impact of Resource Size

The size of a news article can be expressed in many ways: e.g., file size, number of metadata attributes, number of words, number of mentions it contains. Among them, the number of mentions is the property most impacting on the performances of the population process, as each mention must be inserted in HBase and linked to the corresponding news article and entity, and in the KnowledgeStore implementation these operations are more expensive than storing the news article file.

To show the impact of the number of mentions per news article, we conducted a small experiment where we populated the resource and mention layers of the KnowledgeStore under test with two batches of 6000 news articles each from the Cars (Ver. 2) dataset, called ‘small’ and ‘large’ and containing respectively news articles with 44 and 300 mentions (these are the smallest and largest numbers in the dataset). We started with an empty KnowledgeStore and measured the population time and rate for each batch, reporting the results in Table 2.

The difference in performances between the two batches is significant, and partly explains the differences in the population rates measured for the various KnowledgeStore instances of Section 4.2. (see Table 1): excluded the Cars (Ver. 1) instance that used a slower population process, the population rates of the other KnowledgeStore instances inversely correlate with the average numbers of mentions per news article.

5.1.2. Impact of Dataset Size

A degradation of the population rate can be reasonably expected as the amount of data stored in the KnowledgeStore increases and storing and indexing new data become more expensive. We
thus conducted another experiment to assess the extent of this phenomenon and, consequently, the scalability of the population process with large dataset sizes. We populated the resource and mention layers of the KnowledgeStore under test with the whole Cars (Ver. 2) dataset, which consists of ~1.3M news articles (for a total of 2.6M resources, including the corresponding NAF annotated versions) each one having on average 163 mentions (standard deviation = 58). We measured the (instantaneous) population rate at the beginning, in the middle and around the end of the population process, and took note of the state of the system when the three measurements were made, which consists of the number of resources and mentions already stored and the total disk space used in the cluster. The results are reported in Table 3.

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

5.2. Data Retrieval Performances

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

To this purpose, we populated the KnowledgeStore under test with different test datasets of increasing size but similar schema and characteristics, all obtained from the Cars (Ver. 2). We then

Table 2. Population time and rate depending on number of mentions per news article.

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

Table 3. Population rate depending on amount of data already stored. We recall that Resources include both the original news articles as well as its NAF annotated version (i.e., the number of resources is twice the number of news articles)

<table>
<thead>
<tr>
<th>Resources</th>
<th>Data already stored</th>
<th>Disk space [GB]</th>
<th>Population rate [news articles/h]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>7,537</td>
</tr>
<tr>
<td>1,239,270</td>
<td>105,674,491</td>
<td>125</td>
<td>7,872</td>
</tr>
<tr>
<td>2,296,874</td>
<td>189,360,192</td>
<td>224</td>
<td>7,830</td>
</tr>
</tbody>
</table>
selected a set of parametric retrieval requests that are representative of possible interactions of the user with the system for these datasets, starting from the queries of the NewsReader Simple API. We call request mix the instantiation of these requests for a specific set of parameter values and define the evaluation of a request mix as the sequential evaluation of the requests of the mix. For each test dataset and different numbers of clients (independent variables) we simulated the concurrent evaluation by these clients of a large number of request mixes, and we measured the overall request throughput and the average request evaluation time (dependent variables).

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

5.2.1. Test Datasets

We started from the Cars (Ver. 2) dataset due to its real-world contents and large size (~1.3M news articles / ~2.6M resources), and we built five test datasets of increasing size (D1 to D5) by selecting only specific subsets of the source dataset. Table 4 describes the obtained datasets.

The scale factor, computed as the ratio of the numbers of resources in different datasets, provides an indication of the relative dataset sizes. They have been chosen to roughly follow a logarithmic scale, with deviations caused by the practical need to base the selection on the available news article batches forming the source dataset.

5.2.2. Parametric Requests

Table 5 reports the name and informal description of the 14 parametric requests selected for the test (parameters are emphasized in the description).

These parametric requests derive from the ones of the NewsReader Simple API (Hopkinson et al., 2014), an API built on top of the KnowledgeStore to serve practical data analysis needs and used by data analysts and journalists during the Hack Day events (Sections 4.4., 4.5). More details on the selection and the full specification of the parametric requests are contained in Appendix A.

Table 4. The test datasets (from smaller to larger, in terms of number of resources)

<table>
<thead>
<tr>
<th></th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale factor</td>
<td>1.0</td>
<td>2.1</td>
<td>3.5</td>
<td>7.6</td>
<td>15.4</td>
</tr>
<tr>
<td>Resources</td>
<td>163,752</td>
<td>341,464</td>
<td>572,690</td>
<td>1,239,270</td>
<td>2,519,496</td>
</tr>
<tr>
<td>News Articles</td>
<td>81,876</td>
<td>170,732</td>
<td>286,345</td>
<td>619,635</td>
<td>1,259,748</td>
</tr>
<tr>
<td>NAF Documents</td>
<td>81,876</td>
<td>170,732</td>
<td>286,345</td>
<td>619,635</td>
<td>1,259,748</td>
</tr>
<tr>
<td>Mentions</td>
<td>14,031,629</td>
<td>28,836,259</td>
<td>48,329,826</td>
<td>105,674,491</td>
<td>205,114,711</td>
</tr>
<tr>
<td>Entities</td>
<td>2,052,664</td>
<td>4,159,978</td>
<td>6,920,586</td>
<td>14,572,870</td>
<td>27,123,724</td>
</tr>
<tr>
<td>Events</td>
<td>1,829,866</td>
<td>3,752,010</td>
<td>6,285,449</td>
<td>13,398,806</td>
<td>25,156,574</td>
</tr>
<tr>
<td>Persons</td>
<td>81,265</td>
<td>152,951</td>
<td>241,774</td>
<td>446,356</td>
<td>729,797</td>
</tr>
<tr>
<td>Organizations</td>
<td>94,693</td>
<td>177,171</td>
<td>281,526</td>
<td>540,077</td>
<td>947,262</td>
</tr>
<tr>
<td>Locations</td>
<td>46,840</td>
<td>77,846</td>
<td>111,837</td>
<td>187,631</td>
<td>290,091</td>
</tr>
<tr>
<td>Axioms (Triples)</td>
<td>128,035,674</td>
<td>159,180,410</td>
<td>201,090,830</td>
<td>322,981,854</td>
<td>535,035,576</td>
</tr>
<tr>
<td>from Mentions</td>
<td>32,060,740</td>
<td>63,205,476</td>
<td>105,115,896</td>
<td>227,006,920</td>
<td>439,060,642</td>
</tr>
<tr>
<td>from Background Knowledge</td>
<td>95,974,934</td>
<td>95,974,934</td>
<td>95,974,934</td>
<td>95,974,934</td>
<td>95,974,934</td>
</tr>
<tr>
<td>Total Disk Space (GB)</td>
<td>30</td>
<td>50</td>
<td>76</td>
<td>152</td>
<td>280</td>
</tr>
</tbody>
</table>

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
Overall, the parametric requests and their sequential evaluation within a request mix simulate the typical activities of a user exploring the dataset:

- The user searches for events based on certain properties, such as event year, type, term, involved actor(s) URIs, and actor types (requests `sparql 4` to `sparql 8`); in order to constrain these properties, the user may have first to search for a specific actor (`sparql 1`, `sparql 2`) or get an idea of what event terms are in the dataset (`sparql 3`);
- The user then selects an event and retrieves all the information about it (`sparql 9`);
- The user selects an event actor and gets the corresponding description (`sparql 10`), including other persons related to the actor (`sparql 11`) and all the events the actor participates in (`sparql 12`);
- The user chooses a resource mentioning the selected event and retrieves its text (`crud 2`), metadata and mentions (`crud 1`), i.e., the information needed to build a visualization such as the one of Figure 6a.

### 5.2.3. Test Procedure

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

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sparql 1</code></td>
<td>Get first 20 DBpedia actors with given type and name, sorted by number of participated events.</td>
</tr>
<tr>
<td><code>sparql 2</code></td>
<td>Get value of selected property for first 20 actors with given type and name, sorted by value.</td>
</tr>
<tr>
<td><code>sparql 3</code></td>
<td>Get event labels matching a term with number of associated events, in decreasing order.</td>
</tr>
<tr>
<td><code>sparql 4</code></td>
<td>Get summaries of first 20 events having specific year and actor, sorted by event date.</td>
</tr>
<tr>
<td><code>sparql 5</code></td>
<td>Get summaries of first 20 events having specific year and pair of actors, sorted by event date.</td>
</tr>
<tr>
<td><code>sparql 6</code></td>
<td>Get summaries of first 20 events having specific year and actor type, sorted by event date.</td>
</tr>
<tr>
<td><code>sparql 7</code></td>
<td>Get summaries of first 20 events having specific year and label term, sorted by event date.</td>
</tr>
<tr>
<td><code>sparql 8</code></td>
<td>Get summaries of first 20 events having specific year, type and label term, sorted by event date.</td>
</tr>
<tr>
<td><code>sparql 9</code></td>
<td>Get all the information available for a given event (incl. mention URIs).</td>
</tr>
<tr>
<td><code>sparql 10</code></td>
<td>Get all the information available for a given actor (incl. mention URIs).</td>
</tr>
<tr>
<td><code>sparql 11</code></td>
<td>Get first 20 persons related to a given actor, sorted by number of events in common (at least one).</td>
</tr>
<tr>
<td><code>sparql 12</code></td>
<td>Get details of first 20 events participated by a given actor, sorted by event URI.</td>
</tr>
<tr>
<td><code>crud 1</code></td>
<td>Get the metadata of a given resource and the attributes of all the mentions it contains.</td>
</tr>
<tr>
<td><code>crud 2</code></td>
<td>Download the textual content of a given resource.</td>
</tr>
</tbody>
</table>
6. The tests with 2, 3, 4, 6, 8, 12, 16, 24, 32, 48 and 64 clients are run sequentially, 30 minutes each.

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

5.2.4. Test Results

The two line charts of Figure 8 show respectively the throughput measured in request mixes per hour (a), and the average request evaluation time (b) as functions of the number of concurrent clients, with a line for each test dataset.

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

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

The bar chart of Figure 9 shows, for each parametric request, the average evaluation times for the different test datasets.

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

Figure 9. Average evaluation time of each parametric request using one client and varying dataset sizes
6. DISCUSSION

More than a year of concrete usage in NewsReader (including three Hackathon events with users not involved in the project), as well as the experiments reported in Section 5, have provided us with valuable insight on the practical issues and the user expectations encountered when deploying a system like the KnowledgeStore, permitting us to validate our design and identify its weaknesses. In this section we discuss our findings, that we believe are of general interest for any system addressing the same goals as the KnowledgeStore.

6.1. Population Rate

Population rates in the order of few thousands of news articles per hour (Section 4.2.) may appear inadequate, although it must be noted that they comprise the indexing of the news text, its NLP annotations and several hundreds of mentions on average, for a total of several MBs of data. Although we are constantly working towards improving the population rate, the processing required to produce the NLP annotations and to extract mentions is sensibly slower and makes the population cost negligible.51

6.2. Read/Write Separation

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

6.3. Unified Query Language

As motivated in Section 3.1., the choice of a triplestore and SPARQL for managing (schema-free) entity data and of a more-scalable store with CRUD operations and a simpler query language (basically, arbitrarily-complex filtering conditions) for managing resource and mention data appears natural for a system like the KnowledgeStore, as it brings a number of benefits in terms of scalability and compatibility with Semantic Web best practices. Concrete usage of the system shows that users appreciate the expressivity of SPARQL but, deeming inadequate the query language of the CRUD endpoint, they ask for a unified, SPARQL-like language targeted at all the contents of the KnowledgeStore. Providing such a language is however a challenging task due to the volume of data and the different storage backends involved. While we are investigating this direction, we are also considering alternative options, such as: (i) complement (or even replace) HBase with a cluster of independent triplestores on top of which we could deploy federated SPARQL queries; and (ii) adopt a single, distributed graph database as a unified storage backend for all the data managed by the KnowledgeStore, replacing SPARQL with the query language of that database. The latter option, however, would mark a partial departure from Semantic Web standards and its implications and user acceptance have to be carefully evaluated.
6.4. Analytical Queries

Contrarily to our expectations, the KnowledgeStore logs show the submission of many analytical SPARQL queries that match and/or extract a sensible amount of data stored in the KnowledgeStore. It turns out that users submit SPARQL queries to compute statistics, to analyze the loaded corpus and to assess the results and performance of information extraction processors; as discussed in Section 5.2., also the NewsReader Simple API contains some queries that can be classified as analytical. While SPARQL can be used to a certain extent for these investigations, some analytical queries take long times to execute, in some cases due to improper query planning but most often due to their inherent complexity. While we improved some of these queries on an ad-hoc basis, e.g., via careful rewriting (to help the query planner) or by materializing some properties that help speed up queries, and although there are also many analytical queries whose evaluation times (few seconds) are compatible with the online use of the system, a more general and principled approach to handle analytical requests is clearly needed in the KnowledgeStore. A solution we are investigating is the adoption of a high level data manipulation language that can be implemented on top of MapReduce, such as the procedural data-flow language Pig Latin or the declarative, SQL-based language HiveQL. This language can be given to users for formulating their analysis, which can be possibly registered in the system and computed on a periodic basis or when data change, so that their results are always readily available.

6.5. Flexible Access Control

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

6.6. Tighter Integration with Information Extraction Pipeline

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

While a system like the KnowledgeStore should be designed with massive scalability and deployment on a distributed infrastructure in mind, in practice we encountered a number of usage scenarios that do not require scalability and instead mandate for simple, lightweight single-machine deployments; these scenarios include the use of the system for evaluation or demonstration purposes and any other use involving small datasets. It must be noted that the storage backends required in the two deployment situations are very different. A setup for massive scalability employs distributed software infrastructures (e.g., HBase) with substantial overheads that require multiple machines to be competitive, whereas a lightweight setup uses simpler components (e.g., an embedded relational database) that cannot handle large data sizes. Therefore, a flexible architecture with alternative, pluggable storage backends is crucial to enable the system to scale up but also to scale down and be used in a multitude of deployment scenarios. We fortunately chose an extensible, plugin-based architecture for the KnowledgeStore, and we are currently implementing lightweight replacements of the various storage plugins to enable single-process, single-machine deployments of the KnowledgeStore.

7. RELATED WORK

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

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

Apache Stanbol (Gönül & Sinaci, 2012), originated in the IKS Project, is a modular server exposing a configurable set of ReST services for the enhancement of unstructured textual contents. Stanbol main goal is to complement existing CMSs with semantic annotation, indexing and retrieval functionalities. CMS documents and their metadata are fed to the Stanbol server,
where a pipeline of content enhancers is applied to extract entities and additional metadata (e.g., language, topics). Extracted data are augmented with LOD data, and the result is indexed inside Stanbol in a triplestore (similar to the KnowledgeStore) as well as in a SOLR full-text index, supporting respectively SPARQL queries and keyword search. While the KnowledgeStore provides a scalable and reliable primary storage for resources, Stanbol is mainly focused on their indexing for search purposes, and thus their main storage remains in external CMSs. The Linked Media Framework (LMF, Kurz et al., 2014) offers storage and retrieval functionalities for multimedia contents annotated with LOD data. Annotations are provided by external content enhancers such as Stanbol, while the focus of the LMF is on storage and retrieval services as in the KnowledgeStore. Similarly to Stanbol, the LMF data server is based on a triplestore (Sesame) storing annotations as RDF triples and on a SOLR full-text index storing document texts as well as selected metadata and annotation values chosen via XPath-like LDPath expressions; the two storages enable respectively SPARQL queries and keyword-based document search. Similarly to the KnowledgeStore, a ReST API extending the Linked Data HTTP publishing scheme allows read/write access to stored contents. Compared to the KnowledgeStore, KIM, Stanbol and LMF all adopt a ‘two-layer’ model consisting only of resources (text and metadata indexed in a full-text index) and entities (triples indexed in a triplestore). Indeed, storing and querying mention attributes is not a goal of these frameworks. Although mention data could be stored as additional attributes of resources and/or entities, this is not the intended use of these layers and this expedient may lead to inefficiencies or it may be not feasible at all. On the other hand, using the KnowledgeStore as a two-layer system is possible too, but with a small overhead imposed by the unused Mention layer. Therefore, a fair quantitative comparison between the KnowledgeStore and these frameworks is not possible, as they provide different feature sets and they target different usage scenarios. Beyond the different number of layers, another distinctive feature of the KnowledgeStore compared to KIM, Stanbol and LMF is its use of named graph to track the provenance of entities and axioms and to qualify the context where a particular axiom holds.

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

Relevant for our work is also the contribution presented in Croset, Grabmüller, Li, Kavaliauskas, and Rebholz-Schuhmann (2010). The authors present a framework, based on a RDF triplestore, that enables querying the bioinformatics scientific literature and structured resources at the same time, for evidence of genetic causes, such as drug targets and disease involvement. Differently from our approach, this work does not support storing unstructured content (triplestores currently provide only a limited support for integrating knowledge with unstructured resources, often consisting in simple full text search capabilities on RDF literals), and the framework is focused only on specific types of named entities appearing in the unstructured content, whereas a rich, unconstrained set of entities and mentions can be managed in the KnowledgeStore. Another relevant work, in the biomedical domain, is Semantic Medline, a web application that summarizes MEDLINE citations returned by a PubMed search. Natural language processing is performed to extract semantic predications (the equivalent of entity axioms in KnowledgeStore terminology) from titles and abstracts. However, differently from the KnowledgeStore, Semantic Medline has a fixed domain-specific data model, built tailored on that application, and predications can be effectively navigated only on a reasonably small selection of citations (max 500 on the web site)
with no possibility to perform structured queries on the whole corpus (to this respect, a global
index of predications seems missing). Furthermore, while capable of handling large quantity of
resources (21M Medline citations, see Jonnalagadda et al., 2012) the semantic content extracted
and to be handled is proportionally rather small (~57.6M predications of 26 types [32]; cf. with
Cars (Ver. 2) KnowledgeStore instance, with 535M triples from 1.3M news articles).

Although exploited in a different context, dealing with much smaller quantity of content,
also semantic desktop applications such as MOSE (Xiao & Cruz, 2006) and Nepomuk68 are partly
related with the contribution here presented. Semantic desktop applications enrich documents
archived on the personal PC of a user with annotations coming from ontologies. However, an-
notations 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.

8. CONCLUSION

In this paper we described the KnowledgeStore, a scalable, fault-tolerant, and Semantic Web
grounded open-source storage system for interlinking structured and unstructured data. Besides
presenting its design, functionalities and implementation, we reported (i) on its concrete usage
in three application scenarios within the NewsReader EU project, exploiting four datasets of
increasing size (from 18K to 1.3M news articles), as well as (ii) on a number of experiments
measuring the data population and data retrieval performances of the system, demonstrating the
appropriateness and adequateness of the KnowledgeStore to cope with the goals it was designed
for. We also discussed several practical issues and lessons learned from our experience.

Our research and development agenda is already focused on tackling the open issues that
emerged from the experience matured in developing, deploying, populating, and exploiting the
KnowledgeStore in concrete scenarios: among them, the possibility to offer a unified query
mechanism, enabling to submit expressive retrieval requests spanning the whole KnowledgeS-
tore content; the provision of a dedicated mechanism for analytical queries; and the offering of
a scaled down version of the system, able to handle limited-size datasets on small deployment
environments (e.g., an average PC).

ACKNOWLEDGMENT

The research leading to this paper was supported by the European Union’s 7th Framework Pro-
gramme via the NewsReader Project (ICT-316404). The authors would like to thank all those
who contributed to the implementation of the various versions of the KnowledgeStore, including
Alessio Palmero Aprosio, Mohammad Qwaider, Renato Marroquin, Christian Girardi, Michele
Mostarda, and Marco Amadori.

REFERENCES

Proceedings of 6th International Conference on Knowledge Management and Knowledge Technologies
(I-KNOW’06). Springer.


ENDNOTES

1 For instance, see the fast growing rate of the LOD cloud (http://youtu.be/TFXYSwUEOow), or the number of datasets currently available on DataHub http://datahub.io
2 As observed in Gantz & Reinsel (2011), unstructured data still accounted for more than 90% of the digital universe in 2011.
3 http://dbpedia.org
4 http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/
5 http://www.livememories.org/
6 Facts being a special case
8 As will be discussed in Section 3.1., the KnowledgeStore adopts a flexible data model, enabling to represent arbitrary mention attributes according to the specific requirements of the deployment context. As a limit case, a KnowledgeStore instance can be configured with no mention attributes (beside the mention URI), thus making it assimilable, in this configuration, to a 2-layer documents/entities framework. As storing mention attributes has an impact (e.g., in terms of disk space – cf. row “Mention Layer” Disk Space of Table 1, in Section 8), the decision on which attributes to store

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
should be carefully considered based on the requirements of the applications accessing the KnowledgeStore instance.

Released under the terms of the Apache License, Version 2.0.

http://knowledgestore.fbk.eu/
https://knowledgestore2.fbk.eu/nwr/wikinews/
http://knowledgestore.fbk.eu/ontologies/knowledgestore.html
http://www.semanticdesktop.org/ontologies/nie/
http://www.semanticdesktop.org/ontologies/nfo/
http://groundedannotationframework.org/files/2014/01/

While UML class diagrams can only approximate OWL ontologies, they are frequently used to graphically overview their content.

By provenance, here we intend the possibility of linking an extracted entity/axiom and the source (a news article authored by someone) where it occurs. Although other vocabularies could have been used as well (e.g., PROV – http://www.w3.org/TR/prov-o/), we considered the use of GAF (Fokkens et al., 2014) —a specialized vocabulary for linking entities/facts to the mentions denoting them— and Dublin Core Terms —a standard vocabulary for expressing document metadata (e.g., their authors)— the most appropriate choice in this context.

As the characterization of entities is performed through axioms, there is no provision for specific entity subclasses, attributes or relations.

For example, RDFox exhibits a RAM consumption per triple of ~30-60 B depending on the dataset (Motik et al., 2014). We measured an average of ~2000 triples for a news article and its mentions on the NewsReader datasets, which leads to 60-120 KB of RAM usage per news article, meaning that a powerful machine with hundreds of GB of RAM would only be able to handle reasoning for few millions of news articles.

http://en.wikipedia.org/wiki/ACID
http://www.w3.org/wiki/SPARQL
http://hadoop.apache.org
http://hbase.apache.org

Of course, the distribution of mentions across resources affects the selectivities of certain relations (e.g., gaf:denotedBy) and thus query performances; moreover, managing huge resources is supported in the ReST API but not in the current implementation of the UI.

http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/
http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqbenchmark/results/V7/

As confirmed in our usage experience, several hundreds of mentions can be extracted from a resource, each of them described with tens of attributes. Transformed to RDF triples, this means a few thousands of triples for each resource. As triplestore technology can hardly scale beyond a few billions of triples, storing all the data in a triplestore would limit the overall system scalability to a few millions of documents.

The interested reader can see the KnowledgeStore UI in action in a demo video available at http://youtu.be/YVOQaljLt4.

http://www.newsreader-project.eu/
http://dbpedia.org/

Hence, for each news article, two resources are stored: the original news article, and its fully annotated version. However, as shown in Figure 7, we remark that mentions are only linked to the original news article.

http://nlp2rdf.org/nif-1-0
http://knowledgestore.fbk.eu/ontologies/newsreader.html

As the processing and population of the several KnowledgeStore instances occurred at various stages of the project, different development versions of the processors were used for preparing them.

http://ixa2.si.ehu.es/nrdemo/demo.php
http://ic.vupr.nl/~ruben/vua-eventcoreference.ttl
http://rdfpro.fbk.eu

A collection of background knowledge datasets used for populating the KnowledgeStore (including additional datasets of possible practical interest) is available for download on the KnowledgeStore web site at https://knowledgestore.fbk.eu/download.html.

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

We suspect that possible causes include (but may not be limited to) the better/worse use of caches at various levels of the system and increased synchronization overhead beyond a certain level of concurrency.

To give an example, processing a news article of the Cars (Ver. 2) dataset in the NewsReader NLP Pipeline requires 170s, while populating it in the KnowledgeStore takes 0.5s; the pipeline throughput can be increased using parallelization, but the maximum rate achieved so far—4924 news articles per hour on a cluster of 86 machines—is less than the KnowledgeStore population rates measured on a much smaller cluster.

For instance, we materialized rdfs:isDefinedBy annotations that link each vocabulary term to the URI of the ontology defining it.

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

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


For instance, storing mentions as Entity data in the triplestore may lead to an ‘explosion’ of its size, as noted in Section 3.1.

See https://newsreader.scraperwiki.com/ for the specification of the original Simple API queries.
APPENDIX

We report here the complete specification of the parametric requests used in the data retrieval evaluation of Section 5.2. We start with defining the parameters used, describing their relations and computation. We then specify the 14 parametric requests that were previously informally described in Table 5.

Table 6. Request parameters (shortened URIs point to publicly accessible KnowledgeStore pages)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>The URI of an event in the dataset</td>
<td>(<a href="http://bit.ly/1zkELHj">http://bit.ly/1zkELHj</a>)</td>
</tr>
<tr>
<td>event_term</td>
<td>A term in one of the textual labels of the event</td>
<td>‘hire’</td>
</tr>
<tr>
<td>event_year</td>
<td>The year when the event happened</td>
<td>2011</td>
</tr>
<tr>
<td>event_type</td>
<td>The URI of an ontological type associated to the event</td>
<td>eso.JoiningAnOrganization</td>
</tr>
<tr>
<td>actor</td>
<td>The URI of an actor participating in the event</td>
<td>dbr:Burson-Marsteller</td>
</tr>
<tr>
<td>actor_type</td>
<td>The URI of an ontological type associated to the actor</td>
<td>dbr:Company</td>
</tr>
<tr>
<td>actor_term</td>
<td>A term in one of the labels of the actor (e.g., its name)</td>
<td>‘burson’</td>
</tr>
<tr>
<td>actor_property</td>
<td>The URI of an ontological property defined for the actor</td>
<td>dbr:industry</td>
</tr>
<tr>
<td>actor_related</td>
<td>The URI of a related actor sharing an event with actor</td>
<td>dbr:Facebook</td>
</tr>
<tr>
<td>resource</td>
<td>The URI of a news article in the KnowledgeStore mentioning the event</td>
<td>(<a href="http://bit.ly/1BkEZE7">http://bit.ly/1BkEZE7</a>)</td>
</tr>
</tbody>
</table>

A. Parameter Specification

Table 6 describes the parameters used in the parametric requests.

The parameters are not independent but their values must satisfy some dependencies (indicated in the descriptions in the table) in order for the instantiated requests to properly simulate the behavior of a user exploring the dataset (either directly via the UI or indirectly using some application built on top of the ReST API). Each parameter-value mapping allows us to instantiate the parametric requests and obtain a request mix. Given a dataset, a user-configurable number of mappings can be automatically constructed by evaluating 7 auxiliary queries (available on the KnowledgeStore web site) each one extracting all the admissible values for overlapping subsets of parameters: (event, event_year, resource), (event, event_type), (event, event_term), (event, actor, actor_related), (actor, actor_type), (actor, actor_term), (actor, actor_property). Using the query test generator tool, the resulting tuple sets are then joined and the result is randomly sampled to build the desired number of mappings and thus of request mixes.

B. Request Specification

We report below (in Figure 10 and Figure 11) the specification (SPARQL expression, ReST URL) of the parametric requests selected for the evaluation, highlighting parameters in bold face and indicating for each request the NewsReader Simple API query(ies) it has been derived from. The selection process can be summarized as follows:

- we discarded 3 queries of the Simple API out of 18: query ‘11. Get situation graph’ because subsumed by query ‘5. Get event precis’; query ‘18. Types of actors’ and query ‘9. Get the properties of a type’ because pre-computable offline and thus of little interest for our evaluation;
• of the chosen 15 queries, we aggregated query ‘14. Get events with a specific eso value’ and query ‘15. Get events with a specific framenet value’ because very similar;
• we aggregated query ‘6. Get document metadata’ and query ‘7. Get mention metadata’ in a more comprehensive CRUD operation that retrieves the metadata of a news article and all its mentions;
• we added a file download request (crud 2) that is missing in the Simple API for copyright reasons;
• we set OFFSET to 0 and LIMIT to 20 where used (frequent values during the Hack Day events);
• we finally did some semantics-preserving cleanup of the SPARQL query expressions.
Figure 10.

sparql1 1 → 1. Actors of a specified type

SELECT ?actor (COUNT(DISTINCT ?event) AS ?count) ?comment
WHERE {
  ?event sem:hasActor ?actor.
  ?actor a $[actor_type] $[actor_person]; rdfs:label ?l.
  ?l bfo:contains $[actor_term].
}
GROUP BY ?actor ?comment ORDER BY DESC(?count) LIMIT 20

sparql2 2 → 10. Property of actors of type mentioned in news

SELECT DISTINCT ?actor ?value
WHERE {
  ?event sem:hasActor ?actor.
  ?actor a $[actor_type]; rdfs:label ?l.
  ?l bfo:contains $[actor_term].
}
ORDER BY DESC(?value) LIMIT 20

sparql3 3 → 4. Frequency of event labels in events

SELECT ?event_label (COUNT(DISTINCT ?event) AS ?count)
WHERE {
  ?event_label bfo:contains $[event_term].
}
GROUP BY ?event_label ORDER BY DESC(?count) LIMIT 20

sparql4 4 → 12. Get events mentioning a named actor

SELECT ?event (COUNT(+) AS ?event_size) ?datetime ?event_label
WHERE {
  SELECT DISTINCT ?event ?datetime ?event_label WHERE {
    ?event_label bfo:contains $[event_term].
    ?l owltime:inDateTime ?dt.
  }
  ORDER BY ?datetime LIMIT 20
}
GROUP BY ?event ?datetime ?event_label ORDER BY ?datetime

sparql5 5 → 17. Get events mentioning two named actors

SELECT ?event (COUNT(+) AS ?event_size) ?datetime ?event_label
WHERE {
  SELECT DISTINCT ?event ?datetime ?event_label WHERE {
    ?event_label bfo:contains $[event_term].
    ?l owltime:inDateTime ?dt.
  }
  ORDER BY ?datetime LIMIT 20
}
GROUP BY ?event ?datetime ?event_label ORDER BY ?datetime

sparql6 6 → 13. Get events mentioning a type of actor

SELECT ?event (COUNT (+) AS ?event_size) ?datetime ?actor
WHERE {
  SELECT DISTINCT ?event ?datetime ?actor
  WHERE {
    ?event sem:hasActor ?actor.
    ?actor a $[actor_type] $[actor_person]; rdfs:label ?l.
    ?l owltime:inDateTime ?dt.
  }
  ORDER BY ?datetime LIMIT 20
}
GROUP BY ?event ?datetime ?actor ORDER BY ?datetime

sparql7 7 → 15. Search for events with a specified label

SELECT ?event (COUNT(+)) AS ?event_size ?datetime ?event_label
WHERE {
  SELECT DISTINCT ?event ?datetime ?event_label
  WHERE {
    ?event_label bfo:contains $[event_term].
    ?l owltime:inDateTime ?dt.
  }
  ORDER BY ?datetime LIMIT 20
}
GROUP BY ?event ?datetime ?event_label ORDER BY ?datetime

sparql8 8 → 14. Get events with a specific eso value +
16. Get events with a specific framemnet value

SELECT ?event (COUNT(+) AS ?event_size) ?datetime ?event_label
WHERE {
  SELECT DISTINCT ?event ?datetime ?event_label
  WHERE {
    ?event a sem:Event; $[event_type]; sem:hasTime ?dt.
    ?event_label bfo:contains $[event_term].
    ?l owltime:inDateTime ?dt.
  }
  ORDER BY ?datetime LIMIT 20
}
GROUP BY ?event ?datetime ?event_label ORDER BY ?datetime

sparql9 9 → 5. Get event precise

SELECT DISTINCT ?subject ?predicate ?object ?graph
WHERE {
  ( $[event] eso:hasPreSituation eso:hasPostSituation)
  eso:hasDuringSituation.
  GRAPH ?graph ( ?subject ?predicate ?object. )
  UNION (BIND ($[event] AS ?subject)
    FILTER ( ?predicate = $[sameActor] || $[predicate = $[relatedActor]]
      sem:hasTime ?t; rdfs:label ?event_label.
      ?t owltime:inDateTime ?dt.
    )
    ORDER BY ?datetime LIMIT 20
  )
  GROUP BY ?event ?datetime ?event_label ORDER BY ?datetime
Figure 11.

d10 → 2. Details of URI returned by DESCRIBE query

```
DESCRIBE $[actor]
```

d11 → 8. People sharing an event with a named person

```
SELECT $[actor] AS actor2
    (COUNT(DISTINCT ?event) AS numEvent) ?comment
WHERE
    ?actor2 a dbo:Person.
    FILTER(?actor2 != $[actor])
    OPTIONAL { ?actor2 rdfs:comment ?comment }
GROUP BY ?actor2 ?comment
ORDER BY DESC(?numEvent) LIMIT 20
```

d12 → 3. Details of events involving a specified actor

```
SELECT ?event ?predicate ?object (SAMPLE(?t) AS ?object_type)
WHERE {
    SELECT DISTINCT ?event
    WHERE { ?event sem:hasActor $[actor]. }
    ORDER BY DESC(?event) LIMIT 20
    OPTIONAL { ?object a ?t; FILTER (?t = sem:Act || ?t = sem:Place ||
    ?t = sem:Time || ?t = sem:Event ) }
}
GROUP BY ?event ?predicate ?object
ORDER BY DESC(?event) ?predicate ?object
```

crud 1 → 6. Get document metadata +

7. Get mention metadata

```
http://server:port/resources?id=$[resource]>
```

crud 2 → (not in Simple API due to copyright constraints)

```
http://server:port/files?id=$[resource]>
```