Cultural and historical digital libraries dynamically mined from news archives

Papyrus Query Processing Technical Report
Charalampos Nikolaou, Manolis Koubarakis, Akrivi Katifori
Department of Informatics and Telecommunications, National and Kapodistrian Univeristy of Athens, Panepistimioupolis, Ilissia, 157 84, Athens, Greece
{vivi, charnik, koubarak}@di.uoa.gr
# Table of Contents

List of Figures .......................................................................................................................... 3  
List of Tables ............................................................................................................................ 4  
1.  Introduction .......................................................................................................................... 5  
2.  Data Model and Query Language ....................................................................................... 7  
  2.1.  The RDF Data Model ...................................................................................................... 7  
  2.2.  The RDFS Data Model .................................................................................................... 7  
  2.3.  Temporal RDF .................................................................................................................. 7  
  2.4.  The Data Model of the Papyrus Query Processor .............................................................. 8  
  2.5.  The Query Language of the Papyrus Query Processor ..................................................... 8  
3.  The Query Processing Algorithm ......................................................................................... 10  
  3.1.1.  Data Structures ........................................................................................................... 10  
  3.1.2.  Keyword Interpretation ............................................................................................... 11  
  3.1.3.  Graph Exploration ....................................................................................................... 12  
  3.1.4.  Query Mapping ........................................................................................................... 13  
4.  The Architecture of the Query Processing Layer ................................................................. 14  
  4.1.  Components .................................................................................................................... 16  
    4.1.1.  Query Processor ......................................................................................................... 16  
    4.1.2.  Indexer ....................................................................................................................... 20  
    4.1.3.  RDFStore Connection Manager ................................................................................... 20  
  4.2.  Keyword Querying in the context of Papyrus ................................................................. 21  
  4.3.  Conclusions .................................................................................................................... 21  
5.  Implementation Details ......................................................................................................... 22  
6.  User Interface ...................................................................................................................... 23  
7.  Conclusions and Future Work ............................................................................................ 26  
8.  References ............................................................................................................................ 27
List of Figures

Figure 1: The Query Processing Module Architecture .................................................................15
Figure 2: Interaction diagram for keyword search over the History Ontology ..........................16
Figure 3: RDF Data Graph .............................................................................................................17
Figure 4: Summary Graph .............................................................................................................18
Figure 5: Augmented Graph ........................................................................................................18
Figure 6: Papyrus Logical Diagram ............................................................................................21
Figure 7: Papyrus main search screen. The user types the keyword “company”. .....................23
Figure 8: Results on the search on the History Ontology ..........................................................24
Figure 9: News items result results ............................................................................................25
List of Tables

Table 1: RDF Triples using a Single Table Schema

19
1. Introduction

This document is the technical report about the prototype of the query processing module of the Papyrus architecture. The first objective is to choose the models, languages and implementation platforms for handling the storage of the Papyrus Knowledge Base which is composed of the History and News ontologies\(^1\). The History Ontology models secondary historical material modelled in a hierarchy of classes important to historical science whereas the News ontology semantically annotates the archived news content. Appropriate mappings define the relations between the two domains, History and News. The second objective is to design and implement algorithms for processing user queries over the History and News ontologies. Finally, the third objective is to design and implement a visualization tool to assist users on viewing/exploring both ontologies.

The first objective of the query processing module was achieved with our work in the first year of Papyrus in cooperation with partner UTR. The RDF(S) \([21, 727]\) model was chosen as the ontology model for representing the History and News ontology. Other options such as OWL \([10]\) were not considered in detail due to their high complexity for reasoning and query evaluation. A special requirement in Papyrus is the evolution of entities over time. For this purpose a representation model for time had to be used. After a detailed review of existing proposals for adding time in RDF \([6, 4, 5, 18, 2]\), it was concluded that the most suitable work for Papyrus is that of Gutierrez, Hurtado and Vaisman \([6, 4, 5]\). This model was adopted for Papyrus in agreement with UTR who has already employed it in their proposed evolution model for entities \([25, 12]\).

Concerning the storage of the ontologies, the minimum requirements for choosing an RDF store were the support of temporal representation and querying and the support of accessing and querying the store remotely (using HTTP requests). Three RDF stores were examined in detail, Sesame\(^2\) \([17]\), Jena\(^3\) \([19]\) and AllegroGraph\(^4\) \([13]\). From these three RDF stores, Sesame and AllegroGraph support remote accessing and querying through HTTP requests. Concerning the temporal dimension, to the best of our knowledge, there is very little support for temporal representation and querying in current RDF stores.

Our choice for RDF store was Sesame. Our choice for Sesame against Jena is justified by the fact that it has a simple and powerful architecture. To this end, it provides two communication interfaces, the SAIL and Repository APIs. The first abstracts the storage and the inference mechanisms of RDF, whereas the second provides a higher level API for querying and managing either a local or a remote repository uniformly. Finally, a very strong feature of Sesame is the support of Sail stacking; some Sails can be stacked on top of other Sails. In this way, all calls for the bottom Sail are first intercepted by the Sails that are on top of it extending its functionality\(^5\). Comparing Sesame to AllegroGraph, Sesame does not impose any limit in the number of the RDF triples that can be stored, whereas the free version of AllegroGraph imposes a limit of one billion. Apart from that, while AllegroGraph has built-in support for temporal query answering, it is implemented only in its RDF Prolog query language and not in its implementation of SPARQL \([3]\), which is the standard way of querying RDF data according to W3C’ s\(^6\) recommendation. Adding to that the fact that every temporal query that can be expressed in AllegroGraph can also be expressed in the SPARQL implementation of Sesame (and Jena) or easily be implemented programmatically, possibly sacrificing some performance since no temporal indices are present, it is justified to use Sesame against AllegroGraph.

---

1 In the rest of this report, ontologies are assured to include schema and instance knowledge.

2 Version 2.2.4

3 Version 2.5.7

4 Version 3.0.1

5 This feature is the one that the LuceneSail software component takes advantage of to provide the indexing of the RDF store transparently. LuceneSail is presented in Section 2.1.2.

6 [http://www.w3.org/](http://www.w3.org/)
This report refers to the design and implementation of the query processing layer. According to the user requirements (D2.2 User Requirements), the most popular option for querying the Papyrus knowledge base is through the use of keywords. Other alternatives that were examined and dropped are the use of SPARQL query language, form-based query tools, and natural language queries. The reason for not using SPARQL is its complexity that makes it unusable by an non-expert end user. Form-based query tools provide an indirect and supervised way of querying the knowledge base, which may be restrictive to certain types of queries that cannot meet the user needs. As for the use of natural language, it was perceived with scepticism by the end users (i.e., historians). Finally, the use of keywords for querying the knowledge base is a simple, straightforward and flexible way to describe user needs.

In the area of keyword-based querying of RDF data, some interesting work has been done recently [27, 24, 23]. Our work is based on and extends the approach presented in [23]. In [23], the authors present an approach for keyword search on graph-structured data, and particularly RDF. Concepts from the field of Information Retrieval are employed to support an imprecise matching that incorporates syntactic and semantic similarities between a user keyword and the content of the queried RDF data. As a result, the user does not need to know the labels of the data elements when doing keyword search. From the perspective of query answering, the user keywords are interpreted as elements of structured queries letting the user select, in an additional step, one of the top-k computed queries to retrieve all its answers. The authors of [23] have devised a new algorithm for subgraph exploration guaranteeing that the computed results have the $k$ best scores. Last, a strategy for graph summarization is employed that can substantially reduce the search space. In effect, the exploration of subgraphs does not operate on the entire data graph, but a summary one containing only the elements that are necessary to compute the queries.

Our work adopts and improves several of the techniques presented in [23] extending them to meet the requirements of Papyrus. A significant extension is the addition of a temporal dimension to a keyword query (see Subsections 2.4, 2.5 and 3.1.3). To this end, the keyword-based query language is extended with temporal constructs (i.e., "$\text{before} \ 15/05/1985\)" (see Subsection The Query Language), which define temporal constraints on the given keywords. The graph exploration algorithm has been improved appropriately (see Subsection 3.1.3) resulting in better time performance and better quality in the results. An indexing mechanism is employed (see Subsection 4.1.2) which is able to index RDF literals while the algorithm is running as opposed to [23], in which the indexing takes place offline. This is achieved, because the algorithm has been designed to be independent from the storage layer, which can be updated independently. Similar to that is the construction of the graph structures used by the keyword search algorithm which are constructed online without affecting its time performance. Finally, in contrast to [23], all interpretations of a given keyword in the context of the underlying RDF data are taken into account sacrificing performance against better quality of results and satisfaction of the user information needs.

Our deliverable is organized as follows. In Chapter 2, the data model and the query language are presented. In Chapter 3, the algorithm of the Query Processing Layer is presented, which is keyword-based. In Chapter 4, we present the architecture of the query processing layer and describe its components in detail. In the same chapter, we show how this architecture fits with the rest of the Papyrus architecture. Chapter 5 is more technical. There, we present details concerning the implementation of the query processing layer, such as the supported platforms and the tools that were used for the development, along with the software components that depends on. In Chapter 6, the user interface of the Query Processing Layer is presented as it is implemented in the prototype Papyrus platform. Finally, Chapter 7 concludes our work and discusses future work in the context of the project.
2. Data Model and Query Language

In this chapter the data model and the query language of the Query Processing Layer is presented. First, a brief introduction to the Resource Description Framework (RDF) and the Resource Description Framework Schema (RDFS) is given. Second, we give a brief introduction on how RDF(S) has been extended to represent temporal information about resources. Third, we present our data model, which has been built on top of RDF(S) and temporal RDF [6], and finally present the respective query language.

2.1. The RDF Data Model

The data model on which this work has been based is RDF(S). The Resource Description Framework (RDF) is a language for representing information about resources in the World Wide Web. Resources are identified using URIs (Uniform Resource Identifiers) and described using triples. A triple consists of three elements: the subject, the predicate and the object and it is usually written as (subject, property, object), where subject ∈ U ∪ B, predicate ∈ U and object ∈ U ∪ B ∪ L. U stands for the set of URIs, B stands for the set of blank nodes (these are resources that cannot be identified exactly with a specific URI) and L for the set of RDF literals. Using the triple notation, a statement concerning the creator of the web page http://www.example.com can be represented as:


A set of RDF triples is called an RDF graph.

2.2. The RDFS Data Model

RDFS is a language for defining vocabularies that can be used in RDF graphs. These vocabularies specify the classes and properties that can be used in a domain modelled by an RDF graph. Classes and properties are used for describing groups of related resources and relationships between resources. Classes are sets of resources. Elements of a class (nodes in an RDF graph) are known as instances or individuals of that class. To state that a resource is an instance of a class, the property rdf:type may be used. The following are the most important classes in RDF(S): rdfs:Resource, rdfs:Class, rdfs:Literal, rdfs:Datatype, rdf:XMLLiteral, rdf:Property. Properties are binary relations between subject resources and object resources. The built-in properties of RDF(S) are: rdfs:range, rdfs:domain, rdf:type, rdfs:subClassOf and rdfs:subPropertyOf. To avoid distracting the reader, the formal semantics of the RDF Vocabulary are not presented here, except one: all things being described by RDF expressions are called resources, and are considered to be instances of the class rdfs:Resource. The class rdfs:Resource represents the set called “Resources” in the formal model for RDF(S).

2.3. Temporal RDF

The RDF(S) data model fails to capture the different semantics that arise with the description of statements containing temporal information. The first works that proposed temporal features in RDF were by Gutierrez and colleagues [6, 4, 5]. In their proposal, a framework to incorporate valid time in RDF is introduced. Extending the concept of RDF triple, a temporal triple is an RDF triple with an additional temporal label (a natural number). For example, (s, p, o)[t] is a temporal triple which denotes the fact that the triple (s, p, o) is valid at time t. Triples valid at time intervals are then defined by sets of triples valid at time points. Finally, a temporal RDF graph is defined as a set of temporal RDF triples. [6, 4, 5] study the semantics of the proposed extension to RDF, define appropriate query languages for the extension and present results on the complexity of query answering.

---

The representation and reasoning about incomplete temporal information in RDF has also been studied in [4]. The authors of [4] treat indefinite temporal information (i.e., anonymous time) as blank nodes incorporating, also, temporal constraints based on Allen’s interval algebra [16]. The resulting graphs are called c-temporal graphs for which they provide semantics by extending the semantics of temporal RDF graphs defined in their earlier works [6, 5].

From now on and for the rest of this report, we will use the term class to refer to a resource that is an RDF class, the term property to refer to a resource that is an RDF property, the term individual or instance to refer to a resource that is an RDF instance and the term entity to refer to any of these (when there is no reason for differentiation).

### 2.4. The Data Model of the Papyrus Query Processor

In our data model, we have adopted the temporal model and representation of [6, 5] concerning the time during which a statement (i.e., a triple) is valid, but also extended it appropriately in order to associate also a class or an individual with the time during which it is valid, or in other words, define its lifetime. This is done using the semantics of RDF(S) and particularly the fact that every resource is an instance of class rdfs:Resource. Hence, the lifetime of a class $c$ is a temporal triple of the form $\langle c, \text{rdfs:subClassOf}, \text{rdfs:Resource} \rangle [t]$ and in the case of an individual $i$, it is a temporal triple of the form $\langle i, \text{rdf:type}, \text{rdfs:Resource} \rangle [t]$. In contrast to works [6, 5] which deal with definite temporal information, we deal also with indefinite temporal information. Indefinite information concerning a time point can be given as an interval in which the point must lie. In the case of indefinite information about time intervals, the starting and ending time points of an interval are defined in the same way. This way, an interval expression is enough both for time points and intervals (a time point is an interval whose start and end points are the same).

In terms of data representation, a time interval is represented with four natural numbers. These numbers encode the usual data notation year-month-day as done in ISO 8601 encoding\(^8\). The first two numbers denote the interval of the starting point and the last two numbers denote the interval of the ending point. If all numbers are equal, then the interval is a definite time point. If the first two numbers are equal, then the interval has a definite starting time. The same applies to the ending time point. In effect, time intervals may be definite or have indefinite start or end points for which a time interval estimate is known, in which case the information is indefinite.

Formally, a time interval is a quadruple $s_1, s_2, e_1, e_2 \in I$, where $I \subseteq N \times N \times N \times N$ is the set of time intervals and $N$ is the set of natural numbers. An example of such an interval is this: $(19850501, 19850520, 20100301, 20100304)$, which may be used to denote that a statement was valid during the period that started sometime between days 01/05/1985 and 20/05/1985 and ended some time during days 01/03/2010 and 04/03/2010. For readability purposes, such an interval can be given in the form of $\langle 19850501 - 19850520, 20100301 - 20100304 \rangle$. Using this second form, an interval having a definite start time point can be given as $\langle 19850501, 20100301 - 20100304 \rangle$ and an interval having both a definite start and end time point can be given simply as $\langle 198505015, 201003020 \rangle$.

Likewise, a temporal RDF triple is a temporal RDF triple as defined in [6, 5], that is, $s, p, o, t$, where $t \in I$.

### 2.5. The Query Language of the Papyrus Query Processor

The query language, that our work is based on, is keyword-based, i.e., a query is just a set of keywords. To query data with temporal information the keyword query language is extended with temporal constraints. Temporal constraints define a temporal relation between two intervals. Temporal relations may be any of the thirteen temporal relations defined by the work of Allen [1],

---

such as “a before b”, “a meets b”, “a overlaps b”, “a starts b”, “a finishes b”, “a during b” and “a cotemporal b”. The other six can be derived swapping the intervals of the first six relations. All these temporal relations have been implemented as temporal operators on time intervals and extended to take indefinite information into account. In effect, having indefinite time intervals introduces uncertainty in the answer of whether a time interval is related to another in any of these thirteen relations. For example, consider a definite time interval \( a, [1985/05/15, 1985/05/29] \) and an indefinite one \( b, [1984/05/23, 1985/05/10 - 1985/05/25] \). Then, it is uncertain whether “b meets a”, or “b overlaps a” or “b before a”, but it is certain that “a starts b”, “a finishes b”, “a cotemporal b” etc. do not hold. This kind of uncertainty is analogous to the one captured by the uncertainty operator of the work in [20].

Formally, a query is of the form

\[
KL \varphi
\]

, where \( KL \) is a list of keywords and \( \varphi \) is a finite conjunction of formulas of the form

\[
R c
\]

, where \( R \) is one of the thirteen Allen relations and \( c \in I \), i.e., it is a time interval.

Given that a temporal RDF graph in our data model can contain indefinite temporal information, the usual issues known from work on indefinite information in the relational model arise [15]. Thus, a query can have possible answers, certain answers or answers under conditions.

Querying temporal data is orthogonal to the approach taken for keyword querying (which is presented in Chapter 3) in the sense that a temporal constraint given together with a user keyword implies the execution of the keyword search algorithm applying the temporal constraints in every stage of the exploration process. This means that during exploration, an element of the graph model is not explored if does not satisfy the given temporal constraints.

In the context of Papyrus, temporal querying has been designed and implemented, but not integrated in the prototype query processing system to be demonstrated at the 2\(^{nd}\) review of Papyrus.
3. The Query Processing Algorithm

In this chapter, the query processing algorithm employed on the Papyrus platform is presented. This description will prove useful for the understanding of the Query Processing Layer architecture and the interaction of the components that compose it (see Chapter 4).

Our work on query processing is a keyword-based approach which has been based on the work of [23]. We have extended [23] by incorporating in our system the temporal dimension and the concept evolution\(^9\). Let us consider the keyword query "change in Biotechnology 1900-1970" which might be posed by a historian when she researches on issues related to how Biotechnology (as a discipline) changed in the period 1900 – 1970. The semantics of an answer to a keyword query with a temporal dimension, such as the above, is to retrieve related information concerning the keywords, which is valid on the specified time. The temporal dimension in the query can be specified in terms of a date (e.g., “1985/05/15”), a time interval (e.g., “1900 – 1970”), a specific historical period (e.g., medieval) and in relation to all the aforementioned time elements (e.g., “before 1985/05/15”). To this end, all Allen’s temporal relations [1] between time intervals have been implemented and extended to support time with incomplete information (see Chapter 2 for more details). To support concept evolution the exploration process in [23] has been further extended to explore paths that contain information about the way an entity has evolved over time taking account of any temporal constraints in the query.

The process of querying the History Ontology using keywords can be better described with the following phases:

1. **Keyword interpretation**: during this phase each keyword of the query is interpreted as an element of the History Ontology (class, property, individual) and next as an element of the data model that is used to represent the History Ontology. To this end, LuceneSail [11] is used to index all string literals and WordNet [14] to retrieve synonyms for each keyword. This way, we support imprecise syntactic and semantic matching of keywords to RDF entities. Next, these RDF entities are mapped to entities of the used datastructures (see Subsection 3.1.1).

2. **Graph Exploration**: in this phase an exploration of the graph, which is constructed from the schema and the summary of the data graph, takes place. Specifically, the interpreted keywords from the previous stage serve as starting points of the graph exploration. The aim is to find the k best sub-graphs that connect the keywords with each other and satisfy the temporal constraints (i.e., all entities of the subgraphs are valid or have a lifetime that satisfy the temporal relations in the query).

3. **Query Mapping**: during this phase, the sub-graphs of the previous phase are mapped to SPARQL queries which are then evaluated against the RDF store of the History Ontology and the answers are returned to the user.

### 3.1.1. Data Structures

The keyword search algorithm can operate on graph data models representing the underlying ontology. Currently, three different graph models can be used, each one providing a different view on the underlying data and affecting the space and time complexity of the algorithm in a different way. These graph models are named Data Graph, Summary Graph and Augmented Graph using the terminology of [23]. The Data Graph is a view of the underlying RDF graph. The Summary Graph summarizes the Data Graph in the sense that it contains structural (schema) elements only such as classes and properties between classes (see Subsection Query Processor 4.1.1 for details). The Augmented Graph is a super graph of the Summary Graph containing also specific elements of the\(^9\)

\(^9\) In the context of evolution, the term concept has the meaning of the entity as used throughout the whole report.
Data Graph; those that are not present in the Summary Graph and match with a keyword in the query. The keyword search algorithm uses only the last one, that is, Augmented Graph, for the exploration upon submission of a user query and uses the Summary Graph to succinctly represent the underlying data. In effect, a user query corresponds to an Augmented Graph, that is the Summary Graph (the succinct representation of the RDF data) augmented with data that of the Data Graph which are not present on the Summary Graph, but match with the keywords of the user query.

Clearly, the use of Summary and Augmented Graphs plays a significant role on the data and time performance of the algorithm. In the case of data performance, the algorithm uses only the structural information from the Data Graph, which is embedded in the Summary and Augmented Graph which makes the keyword search algorithm capable of working mostly in memory. At this point, it has to be noted that it is absolutely realistic to assume that these graph structures can fit in memory.

For example, as of the November 2009 (i.e., the last snapshot) the DBPedia dataset\(^{10}\) describes 2.9 million “things” [sic] (i.e., resources) with 479 million “facts” [sic] (i.e., triples). From these resources, 205 are classes which are inter-connected with 1200 properties, 1.170.000 are individuals and the rest 479 million resources identify links to external images, web pages and other datasets. Taking into account that DBPedia is a snapshot of Wikipedia\(^{11}\), which is rapidly evolving, mostly in terms of data entry and not in terms of new knowledge that affects its schema, it is realistic to assume that the aforementioned classes and properties, which comprise the Summary Graph, can fit in main memory and that do not incur any significant space overhead to the algorithm.

In the case of time performance, the worst case scenario for the algorithm is to explore the whole Augmented Graph. While this does not incur any significant overhead, it is avoided because it leads to poor quality answers. So, again the time performance is of no concern. The only case in which there is a time performance issue, is when a keyword may have more than one interpretations, that is, when it could correspond to different elements of the Data Graph. In such cases, which are the norm for such search applications, the time overhead is significant, because keyword interpretation becomes a combinatorial problem, since we have to take into account all possible combinations of the interpretations of the user keywords.

More details concerning the construction of these graph models are given in the Chapter 4, where the components of the Query Processing Architecture are described.

3.1.2. Keyword Interpretation

This phase of the algorithm interprets the user keywords to elements of the History Ontology, which can be either a class, a property or an individual. To do that, a keyword index is constructed using the Index component (to be described in Chapter 4) which indexes all literals of the ontology and a WordNet index is employed. For each keyword, first the index of RDF literals is examined taking the matching elements together with the matching elements that were taken from its synonyms using the WordNet index. All these matching elements comprise different interpretations of a single user keyword in the context of the underlying ontology. Each such interpretation, i.e., each element of the Data Graph, is mapped to an element of the Summary Graph. Because of the fact that the Summary Graph contains only structural information, some interpretations will not be mapped to any elements of the Summary Graph. Those elements are added to the Summary Graph, comprising the Augmented Graph. At this time, those elements are named keyword elements. This interpretation process takes place for each user keyword and at a final step all possible interpretation combinations are calculated. It is mentioned that the Augmented Graph is constructed using a single interpretation from every keyword in the user query. In other words, every combination of keyword interpretations is mapped to a different Augmented Graph.

The calculation of all possible interpretation combinations is an issue that is not tackled in keyword search works in general (including the work in \([23]\)) due to its high complexity in time. We believe that this limitation is very restrictive and its support is of great importance due to the fact that the

\(^{10}\) [http://wiki.dbpedia.org/Datasets](http://wiki.dbpedia.org/Datasets)

\(^{11}\) [http://en.wikipedia.org/wiki/Main_Page](http://en.wikipedia.org/wiki/Main_Page)
user has no knowledge about the domain the data represent. Furthermore, using techniques from information retrieval such as stemming, lemmatization, imprecise syntactic matching and synonyms from WordNet, it is evident that the best result of the interpretation process may not satisfy the user needs. In contrast to the work in [23], we calculate all combinations of keyword interpretations and for each such combination a graph exploration is initiated on the respective Augmented Graph. The computation of the top-k answers is done in a naive way. For all top-k answers of each combination the top-k are kept.

From the implementation perspective, each keyword interpretation combination is assigned to a thread from a pool of threads. This way, data structures such as the Summary Graph can be shared among the threads leading to greater performance in space.

### 3.1.3. Graph Exploration

The second phase of the algorithm is the exploration of the Augmented Graph which is the product of keyword interpretation. As previously mentioned, the Augmented Graph is constructed at query time augmenting the information present in the Summary Graph. The exploration process uses the keyword elements derived from the keyword interpretation as its starting elements. Each starting element forms a point of an independent exploration of the Augmented Graph. For this, the concept of *cursor* is employed, which captures a possible path from a keyword element. In effect, during exploration and for each starting element, many different paths are explored and for each of them a different cursor is used. At each step of the exploration process a cursor (i.e., a path) with the lowest cost is chosen for expansion (according to a cost function) from a queue of cursors. If the cursor that is explored has also been explored from other cursors emanating from every other starting elements, then a subgraph has been found which contains information relevant to all starting elements. This subgraph, which is produced by the combination of the paths emanating from all starting elements, is inserted into a list from which only the top-k will be selected in the end. The exploration process terminates when one of the following conditions are true:

1. The exploration depth has reached an upper limit, $d$. In this case all regions with radius $d$ around each starting element have been explored. It is justified that the exploration process be stopped, because it is unlikely to lead to relevant information.

2. All top-k subgraphs have been produced. This happens only when the higher cost of the subgraph of the list of produced subgraphs become less than the lowest cost of newly created subgraphs. Because of the fact that at each step of the exploration, the cursor with the globally lowest cost is selected and expanded, it is certain that paths with the lowest cost are explored at first place. So, new subgraphs will have cost greater than this cost (given that a subgraph is composed of many paths). In the case that this cost becomes greater than the highest cost of all produced subgraphs, the top-k subgraphs are the $k$ subgraphs with the lowest cost.

3. All graph elements have been explored.

The above description of the exploration process is the one that is employed in [23]. We have extended this process in the following ways:

- **Temporal dimension:** the exploration algorithm has been extended to handle temporal data. More specifically, each element of the Data Graph (either edge or node) is associated with a time period denoting its validity time. Upon submission of a user keyword query with a temporal constraint, the exploration process, as described above, expands only those elements that satisfy the temporal constraints. Because of the fact that indefinite temporal information can be queried, elements that possibly satisfy the temporal constraints are visited and expanded, but their contribution to the overall cost is higher, so they are ranked lower in the answer.

- **Exploration termination:** apart from the termination conditions listed above, we have enforced another one that terminates the exploration process of a cursor emanating
from a specific starting element. According to this, when the exploration process is about to expand a cursor to an element that is a starting element itself, it is visited, but a cursor is not produced for that element. The rationale behind this is that if the expansion continued for this element, then the cursors that would be produced beyond that element would have already been produced by the exploration emanating from this starting element itself. This extension is considered significant, because it reduces both the exploration time and space.

- *Synchronous exploration for each combination of keyword interpretation:* this algorithm has been extended to allow synchronous execution of a number of exploration processes sharing a number of resources and data structures.

3.1.4. **Query Mapping**

During the phase of query mapping, the subgraphs that have been produced during the exploration process are mapped to SPARQL queries, which are evaluated on top of the RDF store. This kind of mapping is done based on specific rules that are described in detail in Chapter 4.
4. The Architecture of the Query Processing Layer

In this chapter the architecture of the query processing layer is presented. First we describe the architecture and the role of this layer in a more abstract way and then, in Section 4.1, we describe in detail the various components that the layer is composed of and how they coordinate with each other. Second, we show how this module is integrated in the architecture of Papyrus and the interaction with the other modules.

The query processing module can also function as an independent module on top of an RDF store in other contexts beyond Papyrus. In this sense, the only input it expects is a query given in the form of a list of keywords. The output is a set of RDF subgraphs which are ranked according to how much they satisfy the user information needs as expressed by the query. Put in another way, this module is responsible for doing keyword search over ontologies in RDF(S) format. It is worth noting that in the prototype the user does not realizes the answer as a set of RDF graphs. Instead, before presenting the results to the user, all graphs are unified in a general one and presented using a tree-like structure. This is because end users are familiar with tree-like structures, and in the presence of complicated ones, with a lot of connections, they become confused losing their initial motivation and interest of using the tool. However, the issue of the result presentation is always open to discussion and thinking of novel ways.

In the context of Papyrus, this functionality is applied over the History Ontology and it is then extended in such a way that content from News Ontology, which is relevant to the user needs, can be retrieved. This is done through the exploitation of mappings between the History and News ontologies. These mappings are stored in the Papyrus Knowledge Base and define correspondences between an entity or a conjunction of entities of the History Ontology and an entity or a conjunction of entities of the News Ontology. Details about the semantics and expressivity of these mappings can be found in [26]. From the perspective of the query processing module these mappings are considered simply as entity correspondences. After the user has issued a keyword query and examined the respective results using the prototype Papyrus platform, she is able to select any entity present in the results in order to retrieve relevant news items.

When these mappings are identified, they are used to retrieve the mapped news entities, by taking into account the right part of the retrieved mappings. The next step is to retrieve news items that are related to these specific news ontology entities. Apart from using the exact entities present in the selected mappings, the query processor takes into account the News Ontology structure to produce more relevant results. The News items maybe annotated by either Topics (Papyrus Themes or IPTC News Themes) or Terms (Entities, Concept Evaluations or Slugs). For more information on the Papyrus ontologies, please refer to [22]. When the mapped News Ontology entity is an instance, then all news items annotated with this entity are retrieved. When the entity is a class, apart from news items annotated with that class, also the news items related to the instances of this class are retrieved.

In Figure 1 the architecture of the query processing module is depicted together with the user interaction. The user, who in the context of Papyrus is a historian, submits a list of keywords. The keywords are processed by the Query Processor component and mapped to entities of the History ontology using the index that has been constructed by the Indexer component (see Subsection 4.1.2). These entities are the starting points of the exploration process of the History Ontology. The exploration process computes RDF subgraphs that match the user keywords (see Subsection 3.1.3). Each subgraph is mapped to a SPARQL query (see Subsection 4.1.1) that is evaluated in the RDF store, whose results are used to augment the subgraph. This subgraph is then returned to the user.

At a next step, the user is able to select a number of entities from the subgraphs and issue another query, which with the aid of mappings shall return News Items relevant to the selected entities. Figure 2 shows a form of interaction diagram only for the part of the keyword search over the History ontology which is executed at first place during the previous procedure.
In our approach, IR concepts are adopted (see Subsection 4.1.2 for details) to support imprecise syntactic and semantic matching of keywords to RDF entities. As a result, the user does not need to have knowledge about the context and labels of the underlying ontology when doing keyword search. As we explained earlier, our approach uses the work presented in [23] for keyword searching over RDF graphs implementing several of the design issues and techniques and extending others, such as those that were described in the previous chapter.

In the next section we present the various components of our query processing module in detail.
4.1. Components

The query processing module (Figure 1) consists of three first-level components, the Query Processor (Subsection 4.1.1), the Indexer (Subsection 4.1.2) and the RDFStore Connection Manager (Subsection 4.1.3).

4.1.1. Query Processor

The Query Processor is the main component of the query processing module. This processor receives the input of the query processing module (in the form of a string) and then invokes and controls the rest of the first-level components. The most important part of the Query Processor is the Graph Index which encapsulates the data model of the query module. In fact, Graph Index contains a view of the underlying ontologies which is used for the mapping of the user keywords to the entities of the History ontology and for the exploration of History and News ontology. The Query Processor component uses the Query Parser in order to parse the input and break it in keywords, eliminating duplicates and special characters (such as *, \, ?, "', ~, etc.). These keywords are then mapped to entities of the History ontology by invoking the Indexer component. These entities, which can be nodes or edges of the respective RDF graph of the History ontology, are obtained from the Indexer in the form of URIs. At a second step these URIs are mapped to elements of the data model using the Graph Index producing what the authors of [23] call keyword elements. It is crucial here to draw a distinction between the URIs of the Indexer that refer to URIs of the underlying RDF graph and the respective elements of the URIs in the data model.
Graph Index

The Graph Index embodies the data model of the query module. It defines three types of graphs, named Data Graph, Summary Graph and Augmented Graph. These graphs were presented earlier in Subsection 3.1.1, but here their structure is presented since there is a clear distinction between the type of edges and the type of nodes they use.

**Figure 3: RDF Data Graph**

A Data Graph has three types of nodes: one for entities (entity nodes), one for typed literals (value nodes) and one for classes (class nodes). Furthermore, it has three types of edges, one for properties between entities (relation edges), one for properties between an entity and a typed literal (attribute edge) and one for properties between classes (schema edges). A Summary Graph is composed only of class nodes and schema edges. Apart from the schema edge `subclassOf`, the Summary Graph contains all relation edges of the Data Graph between two entities. To achieve this each such edge of the Data Graph is projected to a schema edge in the Summary Graph between two class nodes that the respective entities belong to. An Augmented Graph is composed of the type of nodes and edges that the Summary Graph is composed of together with value nodes (because a keyword may be matched with a literal) and a special edge (data-schema edge) between a class node and a value node (in order to connect the matched value node with the class node which is the representative of the respective entity that has an attribute edge with the matched value).

Graph Index constructs a Summary Graph view over the History Ontology. The Augmented Graph is constructed at the time of query processing. To achieve that, the Summary Graph is augmented with the keyword elements obtained from both the Indexer and the Graph Index forming the starting points of the exploration process executed by the Explorer component.
Note that the Summary Graph is constructed at the initialization stage of the Query Processor component whereas the Augmented Graph is constructed only when a query is submitted. The exploration process takes place on the Augmented Graph only. The reason behind this is that we are interested in computing queries and not answers to queries; we want to derive the query structure of the computed subgraphs and then have them evaluated over the RDF store. This also leads to a very good performance since the Augmented Graph is much more compact and smaller than the actual Data Graph. Finally, upon construction of these views over the underlying ontologies, the Query Processor does not interact further with the Knowledge Base.

Table 1 shows a collection of RDF triples, whereas Figure 3, Figure 4 and Figure 5 depict the Data, Summary and Augmented Graph for the simple keyword query “2010 koubarakis publications”. Note that nodes representing entities are depicted in a different way than nodes representing classes or typed literals in the Data Graph. Note also that the keyword elements that have been matched with the user keywords are highlighted in the Augmented Graph. These keyword elements are the starting points of the exploration process.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>pro1</td>
<td>type</td>
<td>Project</td>
</tr>
<tr>
<td>pro2</td>
<td>type</td>
<td>Project</td>
</tr>
<tr>
<td>pro1</td>
<td>name</td>
<td>Papyrus</td>
</tr>
<tr>
<td>pub1</td>
<td>type</td>
<td>Publication</td>
</tr>
<tr>
<td>pub1</td>
<td>author</td>
<td>res1</td>
</tr>
<tr>
<td>pub1</td>
<td>author</td>
<td>res2</td>
</tr>
<tr>
<td>pub1</td>
<td>year</td>
<td>2010</td>
</tr>
<tr>
<td>pub2</td>
<td>type</td>
<td>Publication</td>
</tr>
<tr>
<td>res1</td>
<td>type</td>
<td>Researcher</td>
</tr>
<tr>
<td>res2</td>
<td>type</td>
<td>Researcher</td>
</tr>
</tbody>
</table>
Table 1: RDF Triples using a Single Table Schema

<table>
<thead>
<tr>
<th>res1</th>
<th>name</th>
<th>Manolis Koubarakis</th>
</tr>
</thead>
<tbody>
<tr>
<td>res2</td>
<td>name</td>
<td>Yannis Ioannidis</td>
</tr>
<tr>
<td>res1</td>
<td>worksAt</td>
<td>univ1</td>
</tr>
<tr>
<td>res2</td>
<td>worksAt</td>
<td>univ1</td>
</tr>
<tr>
<td>univ1</td>
<td>type</td>
<td>University</td>
</tr>
<tr>
<td>univ1</td>
<td>name</td>
<td>DI&amp;T</td>
</tr>
<tr>
<td>University</td>
<td>subclass</td>
<td>Agent</td>
</tr>
<tr>
<td>Researcher</td>
<td>subclass</td>
<td>Person</td>
</tr>
<tr>
<td>Person</td>
<td>subclass</td>
<td>Agent</td>
</tr>
<tr>
<td>Agent</td>
<td>subclass</td>
<td>Thing</td>
</tr>
<tr>
<td>univ2</td>
<td>type</td>
<td>University</td>
</tr>
</tbody>
</table>

Explorer

The Explorer component is invoked after the keywords have been mapped to elements of the Summary and Data Graph (data model) and the Augmented Graph has been constructed. From these keyword elements, the Augmented Graph is then explored to find a connecting element, i.e., a particular type of a graph element (either edge or node) that is connected to all keyword elements. The paths between the connecting element and a keyword element are combined to construct a matching subgraph. The process continues until the top-k queries have been computed. The resulting subgraphs are given as input to the Query Mapper component of Query Processor in order to construct the SPARQL query that will be evaluated on the RDF store.

After the SPARQL query has been evaluated on the RDF store and the resulting RDF triples have been returned to the user, he can access the content of the News ontology by clicking on resources of these triples. To do that, the Mapper component uses the mappings that are stored in the Knowledge Base to compute the entities that are related to these resources and returns them back.

Query Mapper

The Query Mapper component is responsible for constructing SPARQL queries out of matching subgraphs of the explored Augmented Graph. A complete mapping of such a subgraph to a conjunctive query can be obtained as follows:

- **Processing of graph nodes**: the labels of nodes might be used as constants. Thus, nodes are associated with their labels. Also, nodes might stand for variables. Every such a node is therefore also associated with a distinct variable. To support this two functions are defined, \( \text{constant}(n) \) and \( \text{var}(n) \) that return either the label of a node or a variable.

- **Mapping of relation edges**: relation edges are edges between two nodes that are entities. In the Augmented Graph these nodes denote classes. So, each such edge \( e(n_1, n_2) \) is mapped to three RDF triples of the form:

\[
(\text{var}(n_1), \text{rdf:type}, \text{constant}(n_1))
\]

\[
(\text{var}(n_2), \text{rdf:type}, \text{constant}(n_2))
\]

\[
(\text{var}(n_1), e, \text{var}(n_2))
\]

The first two triples express the fact that \( n_1 \) and \( n_2 \) are instances of the respective classes, while the third triple express the fact that \( e \) is the property that relates these two entities.
• **Mapping of attribute edges**: attribute edges are edges between two nodes of which the first is an individual and the second is a typed literal value. Such an edge \( e(n_1, n_2) \) is mapped to two RDF triples of the form:

\[
(var(n_1), \text{rdf:type}, \text{constant}(n_1))
\]
\[
(var(n_1), e, \text{constant}(n_2))
\]

By traversing the subgraph and by the exhaustive application of these mapping rules, a subgraph can be translated to a query. The query is simply a conjunction of all the triples generated for a given subgraph.

### 4.1.2. Indexer

The Indexer component is responsible for indexing the literals of the History ontology. This index is created and stored together with the Knowledge Base (i.e., the RDF store). The purpose of this index is to facilitate the mapping of user keywords to URIs of the History ontology. For the indexing, the LuceneSail software has been proposed in [11] and is available at [https://dev.nepomuk.semanticdesktop.org/wiki/LuceneSail](https://dev.nepomuk.semanticdesktop.org/wiki/LuceneSail). LuceneSail employs full-text search functionality over RDF by simply combining two well-known established systems: Sesame and Lucene. It employs pure Lucene queries within pure RDF queries (using SPARQL), taking full advantage of the expressiveness of each of them. From the design perspective, LuceneSail can be incorporated into a Sesame system and accessed in a uniform and transparent way. This is achieved without any modifications of the syntax of the SPARQL query language according to [11] and because it has been designed and implemented as a SAIL stack (see Chapter 1). LuceneSail has excellent performance characteristics, while keeping requirements of resources low. It implements many common IR features, which are required in our case, such as:

- stemming and lemmatisation,
- phrase, wildcard, fuzzy, proximity and range queries,
- boolean operators and term boosting

Apart from taking care of the syntactic matching of user keywords our Indexer extracts semantically similar entries such as synonyms from WordNet [14]. Again it uses a component that is offered with Lucene. This component creates a separate Lucene index for looking into the WordNet database. This functionality, while implemented, has not been yet added to the prototype Papyrus query processor to be demonstrated in 2\textsuperscript{nd} review.

### 4.1.3. RDFStore Connection Manager

The RDFStore Connection Manager is the component that takes over the interaction between the rest of the components (i.e., Query Processor and Indexer) and the RDF store (in our case Sesame). Every component that needs access to the RDF Store either for reading the History ontology, the News ontology or the mappings between the two has to invoke an appropriate method of this component. It has to be mentioned here that the RDFStore Connection Manager operates both locally and remotely because uses the HTTP communication protocol for Sesame 2.0 and communicate directly with the Sesame Server\textsuperscript{12}. Currently, the only type of queries that the RDFStore Connection Manager supports and are needed by the query module is SPARQL query evaluation and RDF data loading. Using the first type, the connection manager sends a SPARQL query for evaluation to the repository and returns back the matched resources. With the second type, the connection manager loads RDF data in the repository; the data may come from a file or a URI.

\textsuperscript{12}[http://www.openrdf.org/doc/sesame2/system/ch08.html](http://www.openrdf.org/doc/sesame2/system/ch08.html)
4.2. Keyword Querying in the context of Papyrus

Having described the architecture of the query processing module and the way the components are interacting and operating, it is time to show how this module fits with the general system architecture of Papyrus. The system architecture of Papyrus can be seen in Figure 6 in the form of a logical diagram\textsuperscript{13}. The query processing module that presented in this deliverable can be seen at the bottom of this figure under the name “Query Processor”. The Query Processor receives the user keyword query from the Search GUI module which is simply a module that collects the user keyword query from the text field of a search form. The Query Processor operates as described in Section 4.1.1. The functionality of the Knowledge Base Manager, depicted at the centre of this same figure, is provided by the Sesame RDF store.

\textbf{Figure 6: Papyrus Logical Diagram}

4.3. Conclusions

In this Chapter we presented the architecture of the query processing layer and described the various components that comprise it in detail. Finally, we identified the query processing layer as a module in the architecture of Papyrus and showed how it can be integrated and how it interacts with the other modules of the system.

\textsuperscript{13} D6.1 Papyrus architecture design & implementation plan
5. Implementation Details

This short chapter contains technical details concerning the implementation of the developed query processing module and dependencies to other systems that were utilized.

The query processing module has been implemented in the Java programming language using the Java Development Kit 1.6\(^{14}\). For the development, the Ganymede\(^{15}\) release of the Eclipse IDE\(^{16}\) was (and will be) used. The development took place on a Linux system with the following characteristics: Linux 2.6.28-13-generic #45-Ubuntu SMP i686 GNU/Linux. It has already been tested in a Windows system running the XP Professional operation system.

The query processing module depends on the LuceneSail 1.2.0\(^{17}\) full-text indexing tool of RDF data and Lucene 2.3.2\(^{18}\) search engine. The Papyrus Knowledge Base functionality is offered using the Sesame2 RDF store\(^{19}\). A Sesame Server has been set up at http://pathway.di.uoa.gr:8080/openrdf-sesame/ to provide remote access to the KB with HTTP requests. The Sesame Server provides HTTP access to two repositories, one for the History ontology and one for the News ontology. Their respective repository ids are HISTORY_KB and NEWS_KB. The repositories contain the last version of each ontology in RDF format.

A client can connect and query any of the repositories in two ways:

1. either by using the Sesame package and writing code for connecting to a remote repository,
2. or by communicating with the Sesame Server directly with HTTP requests.

In the first case, a client is able to connect to one of the repositories (for example History) using the following Java code:

```java
Repository myRepository = new HTTPRepository(
    "HISTORY_KB");
myRepository.initialize();
// do something with the myRepository instance
```

In the second case, a client has to follow the HTTP communication protocol for Sesame 2.0\(^{20}\). Finally, the Sesame Server has been deployed in the Apache Tomcat 6.0.20 server\(^{21}\).

---

14 http://java.sun.com/javase/downloads/index.jsp
15 www.eclipse.org/ganymede/
16 www.eclipse.org/
17 https://dev.nepomuk.semanticdesktop.org/wiki/LuceneSail
18 http://lucene.apache.org/
19 www.openrdf.org
20 http://www.openrdf.org/doc/sesame2/system/ch08.html
21 http://tomcat.apache.org/
6. User Interface

As already explained in previous sections, Papyrus project attempts to bridge the news and the history domain. A user of the Papyrus platform (i.e., mainly a historian), who would like to find news information through the use of the vocabulary and terminology of the history domain using keywords, is faced with the screen as show in Figure 7. In this screen there are three options available to the user for searching.

1. **Keyword search** is the simplest option, which looks for the given keywords only in the text of the news items.

2. **Semantic Search** looks in the text but also in the news item metadata produced by the content analysis modules (for details see [8, 9]).

3. **Cross-discipline Search** firstly identifies related entities to the given keywords in the History Ontology and then, through the mappings, retrieves news content. This kind of search is offered by the Query Processing Layer presented in this report.

To demonstrate the function of the Query Processing Layer, let us consider a simple use-case scenario. Suppose that a user would like to find news about companies that are related to Biotechnology.

1. She types the keyword “Company” in the search-box of the screen of Figure 7.

   ![Figure 7: Papyrus main search screen. The user types the keyword “company”.

2. When the user clicks “Search”, a list with history entities appears on the left (Figure 8), which are related with the given keyword conceptually and some syntactically also. The user may

---

22 The domain of Biotechnology is the use-case domain of the Papyrus Platform.
see the entities from the History Ontology related to the keyword “Company”. For specific companies he may see their definition also.

3. At a final step, the user clicks on the entity “Company” and then clicks the search button labeled, “Search”, to retrieve related material from the News Ontology and news content using the stored mappings. Selected entities appear in the “Selected” list on the right of screen shown in Figure 8. In this specific example, the mapping used is the following:

\[ \text{history}_\text{onto}\#\text{Company()} \rightarrow \text{news}_\text{onto}\#\text{Company()} \]

where history_onto and news_onto are the namespaces of the History and News Ontology respectively.

At this point, it has to be mentioned that News items related to companies are retrieved (see Figure 9) even though the keyword “Company” may not be present in the news item text. This is accomplished because the mapping used had at its right side the concept “Company”. As a result all news items containing instances of the concept (individual companies) were retrieved as well.
Figure 9: News items result results.
7. Conclusions and Future Work

This report accompanies the software prototype which is delivered as part of the work for the query processing module of Papyrus. In this report, first, we briefly described the objectives of the module. Then, we summarized the work that has been done until now and then presented the architecture of the query processing layer. The components that make up this layer were introduced and their interaction and operation was described. We also showed the role that this layer plays in the overall Papyrus architecture. Last, we reported the technical details of our implementation concerning the implementation language, the systems on which it was implemented and tested, and the software it depends on.

Our future work will concentrate on the following:

- Evaluating the performance of our keyword search algorithms and implementation. Many relevant experiments have already been done, but due to the fact that they are not yet completed and fully documented, they are not reported yet. For these experiments, different RDF datasets have been used, such as DBPedia\(^{23}\) and DBLP\(^{24}\). Datasets such as IMDB\(^{25}\) and the LUBM\(^{26}\) benchmark have been considered also for future use. Of course, the History Ontology designed and implemented so far in the context of Papyrus is constantly used as a dataset for evaluation as well.
- Improvement of the system functionality according to feedback taken from the actual users of the system, e.g., historians.

\(^{23}\) [http://dbpedia.org/About](http://dbpedia.org/About)
\(^{25}\) [http://www.imdb.com/interfaces](http://www.imdb.com/interfaces)
\(^{26}\) [http://swat.cse.lehigh.edu/projects/lubm/](http://swat.cse.lehigh.edu/projects/lubm/)
8. References


9. D4.4 Techniques for multimodal content analysis.


22. Papyrus Deliverable D3.2 Ontologies for news and historical content.


25. UTR. Deliverable D3.1 Tool set for building and evolving ontologies for news content.
