

QUASAR: Querying Annotation, Structure, and Reasoning

Luying Chen
Oxford University
luying.chen@cs.ox.ac.uk

Michael Benedikt
Oxford University
michael.benedikt@cs.ox.ac.uk

Evgeny Kharlamov
Free University of Bozen-Bolzano
kharlamov@inf.unibz.it

1. INTRODUCTION

An increasing number of systems provide the ability to semantically annotate documents. OpenCalais [4], Evri API [2], Zemanta [6], and AlchemyAPI [1] are web-hosted systems that return *annotated documents*, i.e. documents with annotations that are overlaid on the document structure. Many of the annotations can be linked to standard ontologies, such as DBpedia and YAGO. These annotations give insight as to the meaning of documents in a variety of ways, identifying entities and relationships inside them, classifying them according to topic or theme, and giving the attitude or sentiment of a document or document fragment. In order for users (or applications) to make use of these annotations with a means to access and manipulate documents that contain them, we provide a query language for doing this and demonstrate its utility on a demo system built on top of diverse semantic annotators and external ontologies. We explain how integrating semantic annotations and utilizing external knowledge helps in increasing the quality of query answers over annotated documents by both filtering out irrelevant answers and obtaining extra answers that are not explicitly available in the annotated documents.

The benefit of our querying system for semantically-annotated documents stems from its ability to filter query results based on the presence of annotations in diverse annotation vocabularies, with the filtering specification taking advantage of the multiple kinds of relationships within an annotated document. These relationships include:

Document structure: a query language should be able to ask for annotations that lie in a certain position within a document, within a certain paragraph, etc. E.g. “return all annotations appearing in the first few document’s paragraphs”.

Explicit annotation structure: an annotation may only identify an entity, or may distinguish the particular class to which it belongs. An annotation may even identify the particular entity instance, relating it to a named entity in an ontology. An annotation may be a relationship or role, with its arguments likewise being known or unknown instances.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT 2012, March 26–30, 2012, Berlin, Germany.
Copyright 2012 ACM 978-1-4503-0790-1/12/02 ...\$10.00.

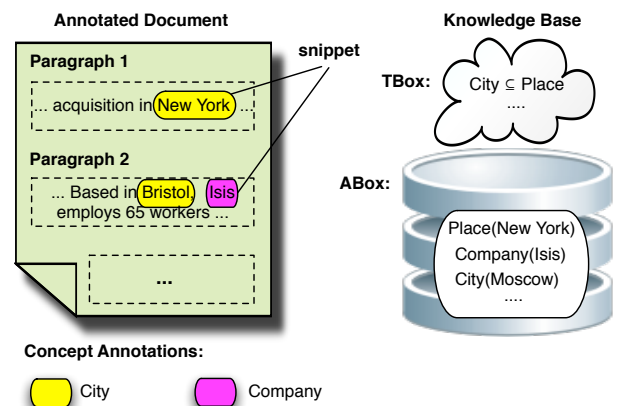


Figure 1: Data Model Example

A query language should be able to ask for all annotations given by a certain annotator, having a certain entity or relationship class, or containing a particular instance. E.g. “return all entities annotated by OpenCalais as a city”.

Implicit knowledge: an important feature of annotations is that they have a well-defined semantics, encoded in the rules of an ontology. A query language should be able to make use of the ontology and exploit *implicit properties* of annotations accessible via reasoning, rather than syntactic matching. E.g., one should be able to ask for all annotations of entities that the ontology infers are politicians – such a query would include entities labeled as presidents, senators, etc. One should be able to ask for all annotations with a person who is known to be married to a US politician – such a query returns snippets containing Hilary Clinton, Michele Obama, etc. Implementing such queries requires integrating *reasoning* with structural querying.

In this paper, we introduce QUASAR (QUerying ANnotation, STRucture AND Reasoning), a system for structured querying of annotated documents that deals with all of the above structural aspects. We start by introducing the data model and QUASAR language in Section 2. In Section 3, we describe the prototype architecture and its implementation. Demonstration details are in Section 4, while the last section discusses the related work. An accompanying video can be downloaded and previewed from URL:<http://www.cs.ox.ac.uk/people/luying.chen/quasar/QuasarDemo.avi>, showing the features of the prototype system.

2. DATA MODEL AND LANGUAGE

2.1 Data Model

We explain the components of the data model using the example in Figure 1. The model has several components:

Document Structure. We assume documents to be divided up into a hierarchy of blocks, which can be referred to and navigated in by queries. In our prototype the hierarchy is of depth three: a document is divided into paragraphs, then into a sequence of sentences, then into tokens. A contiguous sequence of blocks is referred to as a *region*. In Figure 1 we have three paragraphs where the first one has a sentence containing a snippet of two tokens “New York”.

Document Annotations. An *annotation* associates metadata with a region. We refer to the labeled regions as *snippets*. The metadata attached varies from one annotator to another. These semantic annotators could have diverse annotating purposes, such as entity and fact annotation, sentiment annotation and topic annotation. For example, in Figure 1, an entity annotator annotates “New York” with City and “Isis” with Company, along with the URI disambiguating the snippet, e.g. the URI of Isis in DBpedia.

Knowledgebases (KBs). A *KB* consists of a *TBox* or *ontology* which defines structural properties of data, e.g., that every city is a place (in Figure 1 we represent this with a subclass relationship $City \subseteq Place$); and an *ABox* which is a collection of data *assertions* that instantiate classes and binary relations. In Figure 1 we have three ABox assertions, e.g., $Company(Isis)$ which says that Isis is a member of the class Company. We assume that every assertion comes with the URIs of its components, e.g., $Company(Isis)$ comes with the Wiki pages of Isis and of the term company. Note that KBs allow users to query over “extended” facts – facts that could either already exist in the KB, or be inferred by reasoners. In Figure 1 we can infer $Place(Moscow)$.

Using URIs we can link assertions from KBs to annotated snippets of annotated documents, and thus KBs can be employed as external background resources to enhance the quality of query answering over annotated documents. QUASAR has abstract interfaces corresponding to each component of the data model – e.g. for accessing annotators and KBs, and for loading and accessing annotated documents.

2.2 The QUASAR language

We aim for a language that allows querying over annotations, document structure, and information derived from annotations using reasoning. The general form of a query block contains three subclauses: a *SELECT* subclause, a *FROM* subclause and a *WHERE* subclause. More precisely:

```
SELECT   [annotation attributes]   (required)
FROM     [corpora Annotationvars]  (required)
WHERE    [constraints]             (optional)
```

We select annotations FROM a sequence of corpora, associating each annotation with a variable. In the WHERE clause we can impose several kinds of constraints. There are annotation content constraints which can be explicit or implicit, and annotation proximity constraints. Explicit content constraints state *exactly* what predicates or entities should occur in an annotation. We omit details of the grammar due to space limit and illustrate it on the following examples. The following query Q_{place}^1 asks for annotated snippets in the corpus “Corp1” that are annotated as a place:

```
SELECT * FROM Corp1.Annotation ?a
WHERE ?a.assertion.predicate = "Place"
```

Here ?a is an annotation variable. Such a query would be appropriate in a situation where user does not have access to background information. Implicit content constraints state that a predicate or entity occurring in an annotation should satisfy some properties with respect to an ontology. A variant of the query above, Q_{place}^2 allows a user to ask for an annotation in Corp1 that is annotated as a subtype of Place:

```
SELECT * FROM Corp1.Annotation ?a
WHERE ?a.assertion = ?Z(?x) [OntologyFilter:SubType(?Z,"Place")]
```

Here ?Z and ?x are predicate and argument variables, implicitly existentially quantified; OntologyFilter introduces a constraint based on facts inferred from the ontology. In our prototype, the particular ontology being used need not be explicitly referenced in the query above, but is in a separate configuration file. Since determining subtype relationships may require reasoning (they may not be explicit in the ontology), our system has a full reasoner embedded in it.

Annotation Proximity constraints (omitted in the full grammar for brevity) give restrictions on the position of snippets within the document structure. The following simple query Q_{begin} fetches all the annotated snippets in the body of the document occurring in the first two paragraphs:

```
SELECT * FROM Corp1.Annotation ?a
WHERE ?a.snippet.paraNum ≤ 2
```

The QUASAR system allows to combine annotations from different annotators and vocabularies. Consider the query fetching all snippets labeled with Place and mentioned in a negative tone in the document. The relation operator \approx means an overlap between the spans of two snippets.

```
SELECT ?b FROM Corp1.Annotation ?a, Corp1.Annotation ?b
WHERE ?a.assertion.predicate = "Negative"
AND ?b.assertion.predicate = "Place"
AND ?a.snippet ≈ ?b.snippet
```

The flexibility of the language also allows the user to narrow the answers to those that satisfy very specific criteria, thus addressing the problem of “too many answers” that occurs often in keyword querying. As an example, consider a user who is looking for cities which are the birthplace of some politician. The information need can be expressed very specifically with the following query Q_{city} :

```
SELECT * FROM Corp1.Annotation ?a
WHERE ?a.assertion = City(?x) [OntologyFilter:
Birthplace(?y, ?x) AND Politician(?y)]
```

The language also gives the user the ability to control the extent to which implicit information is utilized in a very fined-grained manner. Consider the situation in which a user wants to find annotations mentioning a place. Above we have seen one embodiment of this as a query, Q_{place}^1 . It requires the annotation “Place” to be explicit in the annotation structure but does not require the entity to be recognized as a specific place known to the KB. We have also seen the alternative formulation Q_{place}^2 , in which the annotation can be a subtype of place, but again not requiring the entity to be recognized as a specific place. In contrast, the user could issue the query Q_{place}^3 , asking for annotations that recognize an entity that the ontology knows to be a place:

```
SELECT * FROM Corp1.Annotation ?a
WHERE ?a.assertion = ?Z(?x) [OntologyFilter: Place(?x)]
```

Finally, the user who wants the broadest semantics possible could ask the query Q_{place}^4 , which finds anything that can be inferred to be a place: $Q_{place}^2 \cup Q_{place}^3$.

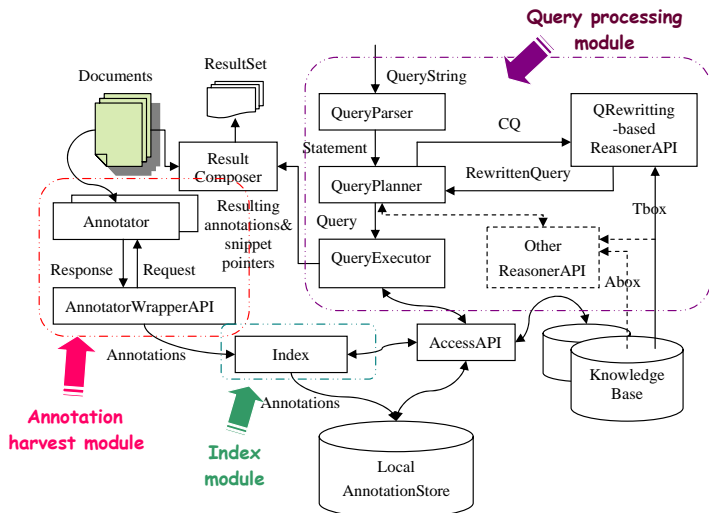


Figure 2: Quasar System Architecture

The ability to get many kinds of “implicit results” addresses the problem of “too few answers” common in traditional search. In summary, the QUASAR language gives the user the ability to combine document structure, explicit annotation structure, and implicit knowledge, while dealing with multiple vocabularies and annotators. It allows the user to pull in implicit results, but allows fine-grained control over how and whether implicit information is used.

3. SYSTEM OVERVIEW

3.1 System Architecture

The architecture of QUASAR is shown in Figure 2.

Offline Modules. The *Annotation Harvest Module* contains two kinds of components: *Annotators* and *AnnotatorWrappers*. In order to integrate annotators with distinct APIs, metadata and response formats, we define a uniform wrapper interface for the interaction between annotators and the system. Based on the interface, concrete *AnnotatorWrappers* must be implemented for each annotator employed. Each *AnnotatorWrapper* submits documents to the corresponding annotator and then harvests the structural information about annotations from the response into the global *AnnotationStore*. Note that the documents themselves remain external to the store; the store use only pointers into those documents for efficiency reasons. The *Index Module* is responsible for maintaining the indices on the explicit information of annotations.

Online Modules. The *Query Processing Module* covers the top-level components for parsing, planing and executing the QUASAR queries, as well as reasoners for inferring implicit information of annotations. The entry point for a QUASAR query is the *QueryParser*. The parsed query is sent to the *QueryPlanner* decides on a strategy for execution, including: which *Reasoner* and *KnowledgeBase* to use (if multiple reasoners/KBs are available), which indices to use for accessing the annotation store. If a *Rewritten-based Reasoner* is used, the *QueryPlanner* interacts with the reasoner to unfold the ontology-related constraints in favor of unions of conjunctive queries over the KBs. Note that the KBs may be accessed remotely (e.g. DBpedia public SPARQL endpoint), the join between structural constraints and semantic constraints have to be performed within *QueryPlanner* it-

self. The *QueryExecutor* fulfills these plans by interacting with the *AnnotationStore* and the *KnowledgeBase*.

The results returned from *QueryExecutor* will generally contain annotation snippets represented by pointers encoding the location within the source documents. The *ResultComposer* translates these pointers into concrete snippets within the document that are suitable for user’s navigation.

3.2 Implementation

The QUASAR system is implemented in Java, with abstract interfaces for the components shown in Figure 2. The choices of these components used in our prototype are:

Annotator. In the prototype, two text analysis APIs – OpenCalais and AlchemyAPI are employed as the semantic annotators. OpenCalais performs entity and relation facts extraction, while AlchemyAPI provide the support for entity and sentiment extraction.

Annotator wrapper. Tailored wrappers for OpenCalais and AlchemyAPI are implemented to harvest the semantic annotations in diverse purposes. JENA RDF API[3] is employed to parse and explore annotation information from RDF-based response of annotators.

Annotation store. Since annotation storage will always be a core component of the QUASAR middleware, we have built an annotation store on top of BerkeleyDB JAVA Edition [8]. It is an embedded non-SQL persistence layer, providing flexible low level access primitives to the annotation objects. Currently we index annotations by annotation predicate and position measured by several granularities – e.g. by document, by paragraph, and by sentence.

Reasoner. Our API allows access to reasoning resources for determining whether a conjunctive query is derivable from a set of facts using axioms of a particular ontology. Because the ontologies we deal with have fairly simple axioms (e.g. DBpedia), currently we are able to use a reasoner based on query-rewriting – REQUIEM [9]. For the ontologies we use, REQUIEM produces a union of conjunctive queries that can be applied to the KB.

Knowledgebases. KBs could either be maintained internally or accessed externally (e.g. endpoints of SPARQL). In our prototype we import the well-known cross-domain ontology – DBpedia as the background KB. We used a standard relational database, MySQL v5.1.51, encoding fact triples as relations. For rewriting-based reasoners, the wrapped format of the rewritten query is thus translated into SQL for evaluation in MySQL. Standard indexing approaches are applied to speed the performance.

Example corpus. The default corpus we use for demonstrating sample queries comes from the *acquisition* subcategory of Reuters-21578[5]. The imported corpus contains 719 documents from Reuters newswire in 1987.

Empirical evaluation. The QUASAR query engine and related persistence layer are set up on Windows XP SP3, Intel Core 2 Quad CPU,2.50 GHz and 3GB of RAM. We tested our system on DBpedia, importing the core ontology ABox of DBpedia 3.5.1, including 5,491,908 and 11,135,755 triples for concept assertions and role assertions respectively. Over 24,000 annotations are extracted from the corpus.

Since the prototype is built upon third-party data storage engines, the query performance is dependent on these components as well as on our own optimization. Here we



Figure 3: Main GUI of QUASAR

give preliminary numbers for our BerkeleyDB/MySQL implementation on the sample queries in Subsection 2.2. Queries Q_{place}^1 and Q_{begin} do not make use of reasoning, and are evaluated with the help of BerkeleyDB indexes: on our sample corpus they take 0.22s and 0.096s respectively. Evaluation time for Q_{city} using a naive query plan requires 4.406s on a hot cache. In fact, further optimization can be done by materializing the ABox of each KB, avoiding the blow-up in query-rewriting. Indeed, DBpedia performs such materialization for the subsumption hierarchy already; taking advantage of this reduces the execution time for Q_{city} to 0.73s.

4. DEMONSTRATION DETAILS

A screenshot of the main components is shown in Figure 3. The demonstration GUI allows a user to either choose a predefined query from a collection of samples or compose a new one from scratch. She can query all the annotated corpora registered in the system’s corpora directory.

The GUI provides users with four modes to preview the set of resulting annotations returned by our query engine: a “Plain list” view, “Group by label” view, “Group by document” view, and “Group by instance” view. For any of these “grouping modes” the group names with results are shown along with the total number of results per group. By clicking on each group entry, the user can further preview the annotation sublist of the corresponding group. Here, each annotation is represented as a highlighted snippet, along with a window giving its context.

From the “Legend” pane, the user can explore the annotation tasks and corresponding vocabularies supported by each annotator. For queries with *OntologyFilter*, the user can browse the results of the rewritten query produced by the reasoner. When the user selects an item from the list of annotations, the system shows the whole text region (the document by default) together with the filtered annotations it contains. The user can navigate the annotations by pressing the *next* or *previous* buttons in the toolbar. If she is interested in one particular highlighted snippet, by mousing over the region she will see a description of all the corresponding annotations associated with the snippet, including annotators, annotation types, and the participating entities from both annotators and knowledge base (if applicable). The user can click the entity URI links to browse more information from external Linked Data and Web assets such as DBpedia and YAGO.

5. CONCLUSION AND RELATED WORK

The QUASAR system is the first step in devising a rich querying environment which enhances structured querying on documents with access to annotation structure and reasoners. We have shown how such systems can allow users to specify their information needs with greater precision. In ongoing work we are exploring enhancements that allow the knowledgebases to be formed dynamically (i.e. as the output of queries), and also a scored semantics that takes into account annotation uncertainty.

Our work comes from the perspective of melding traditional structured querying with ontology and annotation access. In contrast, there has been considerable activity in the DB and IR research communities that takes keyword querying as a starting point and enhances it with some support for “semantics” – e.g. making use of a knowledgebase. Many of these maintain the use of a keyword-based query interface but make use of entity annotations. Others make limited extensions to keyword queries; for example, Ilyas and Pound’s QUICK system [10] provides support for queries that supplement keywords with structured entity/relationship annotations. As with keyword queries, they do not use a “hard” boolean semantics for the language – instead the query processor looks first for entities in a knowledgebase that match the annotations; it then uses these entities to search for relevant documents.

Other work targets enhances keyword search by exploiting the output of semantic annotators. The KIM platform [7] supports access to semantically annotated documents, but with no full query language for accessing the annotation and document structure in tandem. The DOCQS system of Zhou, Cheng, and Chang [11] provides a language for combining keyword search with matching of entities produced by an entity extractor. Their language does not support access to a reasoner. On the other hand, their query languages does support more powerful structure manipulation operations than ours does, such as aggregation and grouping.

Acknowledgements. Benedikt and Kharlamov are supported by EPSRC EP/G004021/1 and EP/H017690/1. Kharlamov is supported by ERC FP7 grant Webdam (n. 226513).

6. REFERENCES

- [1] Alchemyapi www.alchemyapi.com/api/entity/.
- [2] Evriapi. www.evri.com/.
- [3] Jenaapi. jena.sourceforge.net.
- [4] OpenCalais. www.opencalais.com/.
- [5] Reuters-21578. www.daviddlewis.com/resources/testcollections/reuters21578/.
- [6] Zemanta api. <http://developer.zemanta.com/>.
- [7] A. Kiryakov, B. Popov, I. Terziev, D. Manov, and D. Ognyanoff. Semantic annotation, indexing, and retrieval. *J. Web Semantics*, 2(1):49 – 79, 2004.
- [8] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *USENIX Technical Conference*, pages 43–43, 1999.
- [9] H. Pérez-Urbina, I. Horrocks, and B. Motik. Practical aspects of query rewriting for OWL2. In *OWLED*, 09.
- [10] J. Pound, I. F. Ilyas, and G. E. Weddell. Quick: Expressive and flexible search over knowledge bases and text collections. *PVLDB*, 3(2):1573–1576, 2010.
- [11] M. Zhou, T. Cheng, and K. C.-C. Chang. DoCQS: a prototype system for supporting data-oriented content query. In *SIGMOD*, pages 1211–1214, 2010.