Mímir: an Open-Source Semantic Search Framework for Interactive Information Seeking and Discovery

Valentin Tablan, Kalina Bontcheva, Ian Roberts, Hamish Cunningham

University of Sheffield
Department of Computer Science
Regent Court, 211 Portobello
S1 4DP, Sheffield, UK

Abstract

Semantic search is gradually establishing itself as the next generation search paradigm, which meets better a wider range of information needs, as compared to traditional full-text search. At the same time, however, expanding search towards document structure and external, formal knowledge sources (e.g. LOD resources) remains challenging, especially with respect to efficiency, usability, and scalability.

This paper introduces Mímir – an open-source framework for integrated semantic search over text, document structure, linguistic annotations, and formal semantic knowledge. Mímir supports complex structural queries, as well as basic keyword search.

Exploratory search and sense-making are supported through information visualisation interfaces, such as co-occurrence matrices and term clouds. There is also an interactive retrieval interface, where users can save, refine, and analyse the results of a semantic search over time. The more well-studied precision-oriented information seeking searches are also well supported.

The generic and extensible nature of the Mímir platform is demonstrated through three different, real-world applications, one of which required indexing and search over tens of millions of documents and fifty to hundred times as many semantic annotations. Scaling up to over 150 million documents was also accomplished, via index federation and cloud-based deployment.

Keywords: Natural Language Processing, Semantic Search, Scalable Semantic Search Framework, Expressive Semantic Queries, Integrated Semantic Search

1. Introduction

Traditional full-text search is no longer able to address the more complex information seeking behaviour, which has evolved towards sense-making and exploratory search [1]. In the latter cases, traditional precision-oriented approaches from the field of Information Retrieval (IR) are not sufficient. For exploratory search, in particular, recall is paramount, as well as the ability to carry out interactive retrieval [1].

Semantic search over documents aims to address these new challenges by finding information that is not based just on the presence of words, but also on their meaning [2]. It is often referred to as *hybrid* or *semantic full-text search* [3], in order to distinguish it from semantic web search engines, concept search and other types of semantic search (see Section 6 for details). Such systems support hybrid semantic queries, which combine keywords and formal query syntax (e.g. SPARQL [4]), in order to search jointly against document content and ontologies.

Semantic full-text or hybrid search is a modification of classical IR, where documents are retrieved on the basis of relevance to ontology concepts, as well as words. While the basic IR approach considers word stems as tokens, there has been considerable effort towards using word-senses or lexical concepts (see [5, 6]) for indexing and retrieval. In the case of semantic search, what is being indexed is typically a combination of words, formal knowledge typically expressed in an ontology, and semantic annotations mentioning ontological concepts in the text [2].

Natural Language Processing (NLP) is commonly used to derive semantics from unstructured content and to encode it in a structured format, suitable for semantic search. Since some of the most frequently used searches are for persons, locations, organisations, and other named entities [7], some of the most widely used NLP techniques are *named entity recognition* [8, 9], *entity linking or disambiguation* [10], and other types of *semantic annotation* [11].

From a retrieval perspective, entity-annotated content enables semantic search queries such as "LOC earthquake" which would return all documents mentioning a location of an earthquake. Semantic annotation, on the other hand, goes one step further by disambiguating which specific real-world location is mentioned in the text (e.g. Cambridge, UK vs Cambridge, Mass.). Typically a knowledge base or a Linked Open Data (LOD) resource are used as a source of unique entity identifiers (URIs) and formal knowledge about them. This enables even more powerful semantic searches, based on knowledge that is external to document collections. For example, a query on flooding in the UK would retrieve a document about floods in Cambridge, even though the latter does not explicitly mention the UK. The knowledge linking Cambridge to the UK would instead come from, e.g. DBpedia [12] or Geonames[1].

The focus of this paper is on Mímir[2] – an open-source framework for integrated semantic search over text, document structure, linguistic annotations, and formal semantic knowledge.

Typically semantic full-text search approaches enlarge standard IR indexes with semantic terms (e.g. URIs), while still modelling documents as bags of tokens and disregarding their structure. In contrast, the Mímir semantic search framework uses two additional types of data: *linguistic annotations* created by NLP tools (e.g. morphology, part-of-speech, and syntax) and *document structure annotations* (e.g. paragraphs, sections, titles). In order to distinguish this from the bag-of-words-based semantic full-text search approaches, the term *integrated semantic search* is introduced.

The novelty of the Mímir semantic search framework lies in its support for serendipitous information discovery tasks, to complement information seeking searches. Exploratory search and sense-making are supported through a number of visualisations, including co-occurrence matrices and term clouds, as well as an interactive retrieval interface, where users can save, refine, and analyse the results of a semantic search over time. The more well-studied precision-oriented information seeking searches are also supported, including ranking of search results. To the best of our knowledge, the Mímir framework is the first open-source semantic search platform of this kind.

The novel contributions of this paper are:

1. An in-depth description of the Mímir open-source framework, including its architecture (Section 2), the indexing and search over document text, structure, linguistic annotations, and formal semantic knowledge (Sections 2.1 and 2.2 respectively). In particular, direct indexes are created in addition to the widely used inverted indexes, in order to support both information discovery and information seeking searches. Direct indexes power the dynamic calculation of sets of frequently occurring terms within relevant document lists. These term sets underpin user interfaces that support the discovery of new knowledge and relationships by displaying term clouds and co-occurrence matrices as part of interactive retrieval tasks.
2. Presentation of two semantic search interfaces for information seeking tasks from two real-world applications (Section 3.1).
3. Presentation of a semantic search interface for information discovery, including a real-world application in knowledge discovery from immunology literature (Section 3.2). 9.5 million documents are searched interactively, in conjunction with a medical domain ontology.
4. A comprehensive evaluation of the Mímir semantic search framework. Firstly, intrinsic evaluation is carried out, with respect to indexing and search efficiency (Section 5.1). This includes also the evaluation of a complex semantic search query against 9.5 million documents, 3.5 billion tokens, and 743 million linguistic and semantic annotations (Section 5.2). Secondly, extrinsic evaluation is carried out with users, as part of a semantic search application which combines environmental science literature and Linked Open Data (Section 5.3).
5. Positioning Mímir with respect to the state-of-the-art (Section 6), including a detailed comparison against the Broccoli semantic full-text search system, which is its nearest analogue.

2. Mímir: an Open-Source Semantic Search Framework

Mímir[3] is an integrated semantic search framework,

[1]A geographical database available from http://geonames.org.
[2]http://gate.ac.uk/mimir/

[3]Old Norse "The rememberer, the wise one"

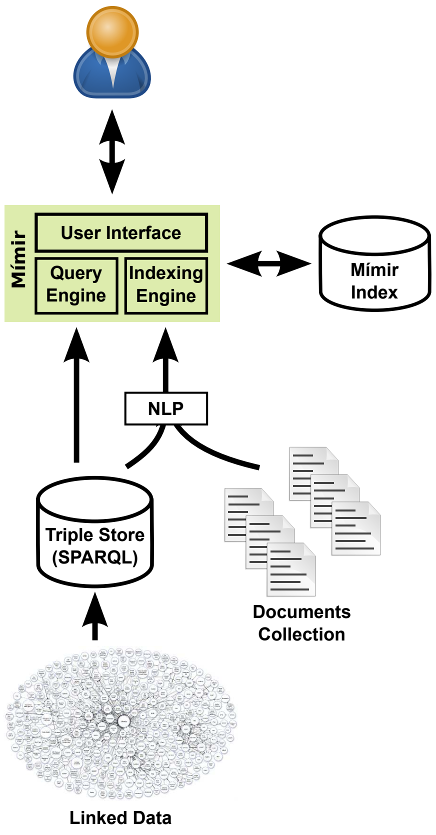


Figure 1: Mímir life cycle.

which offers indexing and search over full text, document structure, document metadata, linguistic annotations, and any linked, external semantic knowledge bases. It supports hybrid queries that arbitrarily mix full-text, structural, linguistic and semantic constraints. A key distinguishing feature are the containment operators, that allow flexible creation and nesting of full-text, structural, and semantic constraints, as well as Mímir's support for interactive knowledge discovery.

Mímir has been designed as a generic and extensible, open source framework[4]. It can also be used as an on-demand, highly scalable semantic search server, running on the GATECloud [13] platform.

The high-level concept behind Mímir is illustrated in Figure 1. First a document collection is processed with NLP algorithms, such as those provided by GATE [14]. Typically the semantic annotations also refer to Linked Open Data resources, accessed via a triple store, such as OWLIM [15] or Sesame [16]. The semantically annotated documents are then indexed in Mímir, together with their full-text content, document metadata, and document structure markup. At search time, the triple store is used as a source of implicit knowledge, to help answer the hybrid searches that combine full-text, structural, and semantic constraints. The latter are formulated using a SPARQL query, executed against the triple store.

Mímir's architecture is shown in Figure 2. It is implemented as a web application that runs server-side and can optionally be distributed across multiple machines. It includes both information seeking and information discovery user interfaces, as well as a REST-style API for programmatic access. Each Mímir instance manages one or more indexes stored locally. Additionally, it can also use the REST APIs to access any index served by remote Mímir instances and make it available locally. Any set of local and remote indexes can be grouped together in a *federated index*.

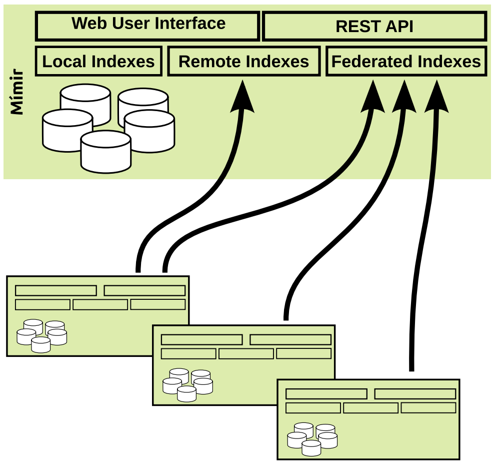


Figure 2: Overall Mímir architecture.

When used in combination with remote indexes, federation enables Mímir to scale to millions of documents, through very large indexes, distributed across multiple servers. Another use case for federation is enlargement of existing indexes with newly indexed content (e.g. new kinds of semantic annotations). Federation works well, since all semantic searches are local to a document. This enables:

- Faster indexing: less content in each Mímir index;
- Faster searches: search space is broken into slices that are searched in parallel.
- Joining search results is trivial: union of result sets.

[4]Download from `http://gate.ac.uk/mimir/`.

$$termID^1 \rightarrow \begin{array}{l} docID^1_1(pos, pos, \ldots pos); \\ docID^1_2(pos, pos, \ldots pos); \\ \ldots \end{array}$$

$$termID^2 \rightarrow \begin{array}{l} docID^2_1(pos, pos, \ldots pos); \\ docID^2_2(pos, pos, \ldots pos); \\ \ldots \end{array}$$

$\ldots$

$$docID^1 \rightarrow \begin{array}{l} termID^1_1(count); \\ termID^1_2(count); \; \ldots \end{array}$$

$$docID^2 \rightarrow \begin{array}{l} termID^2_1(count); \\ termID^2_2(count); \; \ldots \end{array}$$

$\ldots$

Figure 3: Inverted (above) and direct (below) index representations

- Scalability through adding more nodes and creating a federated index.
- Search speed stays almost constant while data increases: each individual repository has the same amount of data; there are just more repositories, federated together.

2.1. Indexing Annotated Documents with Mímir

Mímir uses a compound index approach, where different types of content are stored in separate but aligned *inverted* and *direct* indexes.

Inverted indexes are used to find which of the documents contain occurrences of search terms, where *terms* may be words, annotations, or entity mentions. These indexes are used to support information-seeking tasks where each query is executed to produce a result set comprising documents.

The inverted indexes include position information, which is represented in terms of token offsets. Ignoring implementation details, an inverted index is a list of term IDs and associated *posting lists* (see top half of Figure 3). The posting list for each term is a sequence of postings, each including a document ID and a position. Term and document IDs are long integers and consistent between the corresponding direct and inverted indexes. Term IDs are ordered by the lexicographic order of the term strings; document IDs represent the indexing order of the documents; and all posting lists are sorted by document or term ID.

Mímir also provides *direct indexes*, which map documents to terms. Given a set of documents (typically retrieved in an earlier search), direct indexes are used to find which terms, of any type, occur in those documents, and with what frequency. This functionality is used to support information discovery tasks, which are exploratory in nature. In such cases, users typically require visualisations of the most frequent terms, co-occurrence matrices, and other interfaces for interactive document retrieval and analysis (see Section 3.2).

The use of direct indexes is optional and can be configured when a new index is being built. Direct indexes are smaller than their inverted counterparts, because they are not required to include positional information. The latter is already stored in the corresponding inverted index. The two types of index are illustrated in Figure 3.

The on-disk indexes are implemented using the MG4J library[5][17]; if required, similar indexing structures can be implemented using alternative IR engines, such as Lucene[18].

Conceptually, a document that has been processed with NLP tools comprises textual content and annotations (some encode structural markup, whereas other represent automatically created syntactic and semantic information). Additionally, document metadata may also be available (e.g. mime type, creation date). Mímir uses the document representation format defined by the GATE framework [14] as an interchange format, since it can represent all these types of information.

The process of indexing a new document is shown in the top part of Figure 4, which we describe in more details next.

2.1.1. Indexing Document Text

Many indexing systems perform some form of pre-processing of the input text prior to indexing, e.g. tokenisation, word stemming, elimination or normalisation of accented characters. Since Mímir is designed to work with NLP pre-processed documents with linguistic annotations, no additional document pre-processing is carried out. There are two advantages to this approach. Firstly, NLP tools tend to be more sophisticated and better at handling different languages and scripts. Secondly, it brings flexibility, since pre-processing is done prior to indexing and can be changed easily. Consequently, Mímir does not index directly document text, but instead assumes the presence of `{Token}` annotations. Even though they are linguistic annotations, tokens are indexed differently from other annotations.

The simplest text token is a word, but tokens are also used to represent symbols, numbers, or punctuation. These are generated automatically by the NLP pre-processing tools, which would also customarily associate some metadata with each token (e.g. orthography, morphological root, part-of-speech). In GATE and

[5]http://mg4j.dsi.unimi.it/

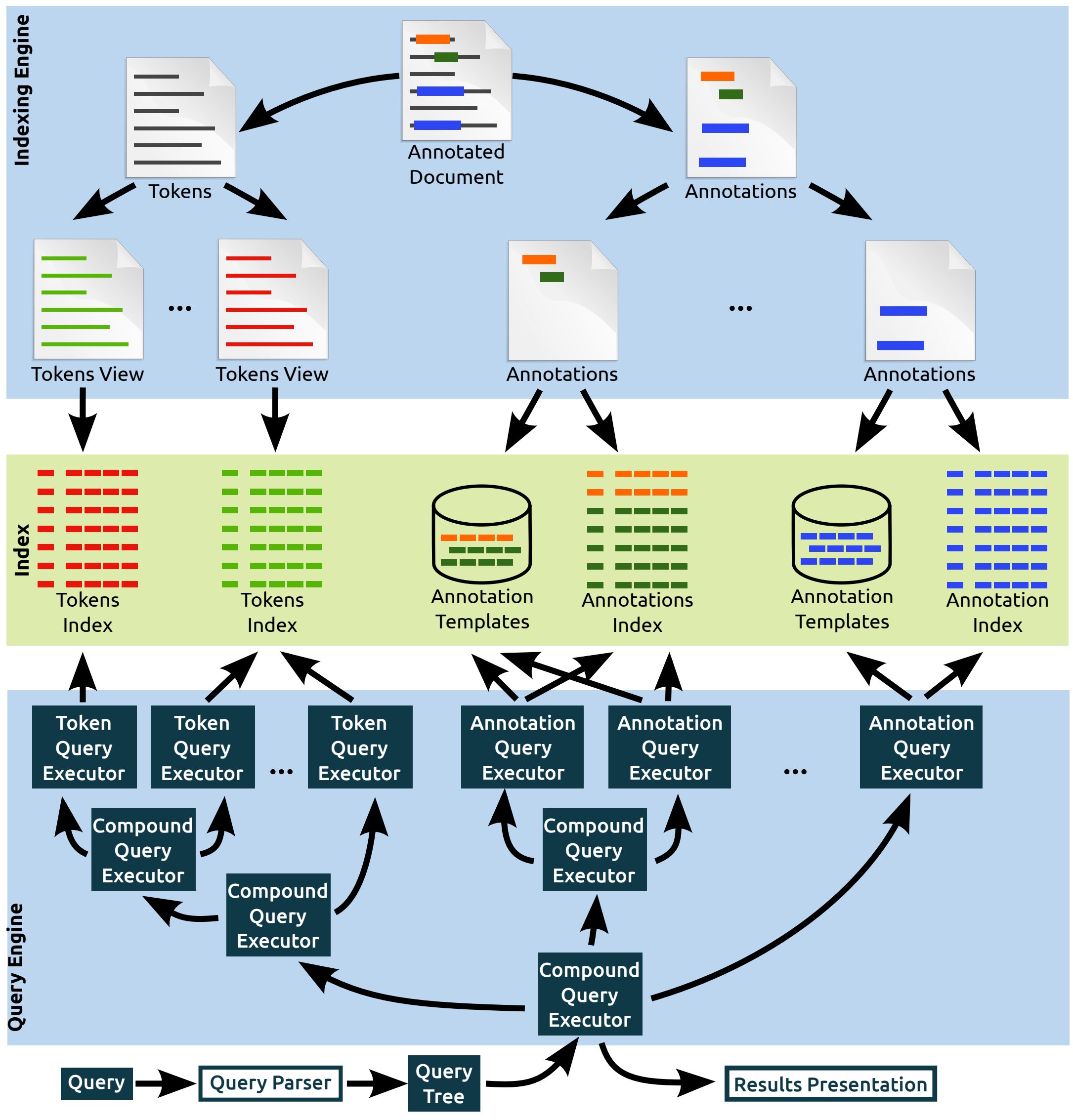


Figure 4: High level view of the data and execution flow when creating and searching a Mímir index.
The upper part of the figure illustrates the indexing process, while the bottom refers to the execution of queries. The central band shows the types of data stored into the compound index.

thus Mímir, these are coded as feature-value pairs associated with the token annotations.

Each token feature, including the `string` can be selected for indexing through configuration options. The values for each indexed feature are used to produce each of the different token sub-indexes, and all sub-indexes are aligned to use the same token positions. Figure 5 shows indexing examples for some types of metadata, typically associated with tokens.

2.1.2. Indexing Annotations

The other main kinds of data are the structural and NLP-generated annotations. In Mímir both kinds are represented in the same data structure, comprising a start and end position, an annotation type (e.g. Location, span, div), and an optional set of features.

For efficiency reasons, the standard GATE annotation model was simplified through the introduction of *strongly typed features*. In the standard GATE annotation model there is no requirement that all permissible

Document:	London	is	located	on	the	Thames	.
position:	0	1	2	3	4	5	6
string:	london	is	located	on	the	thames	.
root:	london	be	locate	on	the	thames	.
part-of-speech:	NNP	VBZ	VBN	IN	DT	NNP	.
Location:	`type=city`					`type=river`	

Token indexes

root index	
.	0(6)
be	0(1)
locate	0(2)
london	0(0)
on	0(3)
thames	0(5)
the	0(4)

PoS index	
.	0(6)
DT	0(4)
IN	0(3)
NNP	0(0, 5)
VBN	0(2)
VBZ	0(1)

Location templates

L1 ID	**type**
1	city
2	river

L2 ID	**L1 ID**	**instURI**
1	1	dbpedia.org/resource/London
2	2	dbpedia.org/resource/Thames_river

Mention ID	**L1 ID**	**L2 ID**	**length**
Location:1	1	-	1
Location:2	1	1	1
Location:3	2	-	1
Location:4	2	2	1
Location:5	2	2	3

Location index

{Location} index	
Location:1	0(0)
Location:2	0(0)
Location:3	0(5)
Location:4	0(5)

Figure 5: A very simple example document and the corresponding contents of a Mímir index. We assume that the only document ID is 0.
Different *views* of the document text are generated by different token features, which are stored in separate sub-indexes. The document string has been down-cased prior to indexing; we do not show the `string` index, as it is very similar to the one for the `root` feature. The values used for Part-of-Speech (PoS) are standard tags as produced by GATE's PoS Tagger: DT=determiner, IN=preposition, NNP=proper noun, VBN=verb - past participle, VBZ=verb - 3rd person singular present.
A single annotation type ({`Location`}) is being indexed, with two different occurrences, and we assume the only non-nominal feature to be the DBpedia instance URI. Note that "`Location:5`" (i.e. a mention of the Thames that is 3-tokens long) does not actually occur in the document text, so it is not present in the index. We have included it here as an example of an annotation of length greater than 1.

annotation types and features are declared explicitly. Instead, documents can have any number of annotation types, and annotations can have any number of features, with arbitrary names. Feature values can be any Java object, and also features with the same name on different annotations can take values of different kinds.

For Mímir indexing however, strong typing is enforced through an index definition file. More specifically, this specifies:

1. which annotation types should be indexed;
2. which features for each annotation type should be included; and
3. the types of values that each feature takes.

The feature types are also limited to `nominal` (i.e. with values from a pre-defined set of permitted string values), `string`, `number`, or `URI`.

The type structure is used to infer *annotation templates*, i.e. tuples of values that describe classes of annotations that encode the same information, as follows.

For each input annotation the following IDs are retrieved (or generated on first occurrence):

Level-1 template ID The annotation type and the values for all its nominal features form a tuple. The first time each tuple configuration is seen, it is allocated a level-1 ID. Subsequent annotations that match an already existing tuple will re-use the same level-1 ID. For example, in Figure 5 all annotations of type *Location* with feature *city* will use the level-1 ID '`1`'.

Level-2 template ID The level-1 template ID together with the values for all the remaining (i.e. non-nominal) features form a second tuple. Unique configurations of these tuples are allocated level-2 IDs. It should be noted that most NLP annotations tend to include only nominal features, so they would not require a level-2 ID. The `{Location}` annotations shown in Figure 5 have a non-nominal feature, so they each get a level-2 ID allocated to them. All further mentions of e.g. the *Thames* would re-use the same IDs, even when phrased differently in the text, e.g "*the river Thames*", or "*La Tamise*".

Mention ID The level-1 ID and the annotation length (number of tokens) forms a tuple, which is associated with a mention ID. In Figure 5 *Location* annotations with feature *city* covering one token will take the mention ID "Location:1". If present, the level-2 ID and the annotation length also get a mention ID. For example, all mentions of "the River Thames" are associated with mention ID "`Location:5`" (because they refer to the Thames and are 3 tokens long).

Finally, the one or two mention IDs associated with each annotation are added to an *annotation index*, using the annotation start token as the position.

We index two separate mention IDs associated with either level-1 or level-2 IDs, in order to speed-up searches that only make use of nominal features. For annotation types that have non-nominal features, the number of level-2 IDs will be orders of magnitude greater than that for level-1. If a search only relies on nominal constraints (a large proportion of searches tend to fall into this category), then the query can be answered much faster by only accessing the smaller number of posting lists for the matching level-1 IDs.

Continuing the Figure 5 example, we see that all annotations of type *Location* with feature *city* can be encoded with a single level-1 ID and a small number of mention IDs, one per length, *regardless* of what other features the original annotation held. For example, a search for all cities will only need to access the tables for level-1 and mention IDs, and then search the index for the term "`Location:1`". Similarly, a search for all locations (regardless of their type), will access the same tables and then search the index for "`Location:1 OR Location:3`". By contrast, a search for mentions of the river Thames will need to access the level-2 ID table and then search for "`Location:4` OR `Location:5`".

Each Mímir sub-index is built using its own dedicated execution thread. The configuration supplied when creating a new compound index allows the user to specify which annotation types should be stored in which sub-index. This enables load balancing, according to the expected relative density for each type of annotation.

Indexing semantic annotations: In cases where documents have pre-existing (e.g. RDFa) or automatically created semantic annotations, these will typically have URIs as values for some of their features. At indexing time, URI feature values are not treated differently. They simply become non-nominal features of type string, which are indexed, as already described above. At search time, however, a semantic knowledge base is required, in order to resolve the indexed URIs, e.g. as a SPARQL end-point (see Section 2.2.4).

Annotation templates: The annotation indexing process needs to store some data that allows it to match value tuples to level-1, level-2, and mention IDs. This data becomes part of the Mímir index, and is labelled as *Annotation Templates* in Figure 4. The Mímir framework allows pluggable implementations for the 'helpers' that match tuples to IDs. The default implementation uses an H2[6] in-process relational database for this. An alternative implementation exists, which uses a Sesame[7] triple-store.

At indexing time, one of the most time-consuming operations is searching the list of known level-1 and level-2 IDs, to check if the feature values for the current annotation have been seen before, or whether new IDs need to be generated. This indexing overhead though pays off at search time: IDs allow us to use the same single representation (the ID) for all input annotations that are identical as described by the set of features being indexed. This results in annotation indexes with a small set of quite dense posting lists, instead of a very large set of very sparse ones. The latter would be significantly more inefficient time- and space-wise.

The default implementation of annotation indexing includes an in-memory cache, which stores the mapping between feature value tuples and IDs. When enough RAM is available for a sufficiently large cache, the indexing process is significantly faster, typically in orders of magnitude. This effectively removes the time penalty associated with retrieving annotation IDs.

2.1.3. Document Metadata

Mímir distinguishes between document metadata that needs to be searchable, and document metadata that needs to be retrieved as part of the search results. For example, some applications require searching for documents published between certain dates. Then once a document is matched, the application would typically obtain the original web URL of that document. In this case, the document date becomes searchable metadata, while the document URL is simply stored metadata.

Mímir offers support for searchable metadata through *document-mode* annotations. These are virtual annotations that are declared in the index configuration but which do not actually exist in the input document. Their position is always 0 (they are modelled as annotations that start on the first token of the document), and no length values are stored. At search time, document-mode annotations behave as if they have the same length as the document that 'contains' them. This allows all search operators to work as if document-mode annotations span the entire document content.

There is also support for storing document metadata that is not searchable, but is available on request. This can take the form of any serialisable Java object and is stored in zip files within the index directory.

[6]`http://www.h2database.com/`

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

2.2. *The Mímir Query Language*

Queries in Mímir are trees, with compound query operators as intermediary nodes and base queries as leaves. The latter address directly one of the sub-indexes. Of the query operators described below, the basic types are `String` and `Annotation`, while the rest are compound.

When executing a query, the first step is to parse the query text and convert it into a *query tree*. For each query operator, the query engine creates a query executor which executes the query and produces hits, if any are found. Compound operators use the hits returned by their constituents to calculate their own hit list. The hits produced by the root executor constitute the results of the entire query. A hit in Mímir is a document snippet identified by a document ID, a start offset (a token index) and a length (the number of tokens). All operators use minimal interval semantics: they always return the shortest hit that satisfies the query. The search workflow is illustrated in the lower part of Figure 4.

2.2.1. *Base Operators*

Base operators in Mímir are those that directly access an index. Currently there are two types of base operator: *string*, which provides full text search, using the token indexes, and *annotation*, which searches based on annotations and their features.

String: a base operator which looks up the query term in one of the token indexes. By default the first token index listed in the configuration is used. Conventionally this is the `string` index. If another index should be used, this can be addressed by prefixing the query with its name, e.g `'root:be'` would match all words for which the morphological root is 'be', whereas `'pos:NNP'` would match all proper names, as detected by the POS tagger.

Annotation: this query is based on an annotation type and a set of constraints over the feature values. For example, the query `{Location type="city"}` matches annotations of type `Location` which have a feature named `type` with value `city`.

The workflow for executing an annotation query is:

1. If the search constraints refer to any non-nominal features, then construct the set of all *level-2* IDs for which the feature value tuples match the provided constraints. Otherwise find the set of *level-1* IDs that match.
2. Retrieve the set of *mention* IDs associated with the list of IDs found in step 1. For each mention ID identified, keep track of the corresponding annotation length.
3. Look up all identified *mention* IDs in the corresponding annotation index and, for each match, return a hit with the appropriate location and length.

Other types of feature constraints are supported beyond simple equality. Comparison operators <, <=, >= and > can be used with numeric features and regular expression matching can be used for strings. A common strategy to handle date expressions, for example, is to normalise their values and encode these as numbers in the form *YYYYMMDD*, supporting searches, such as `{Document date > 20120000 date < 20130000}`, which would find documents from the year 2012.

2.2.2. *Standard Compound Operators*

AND(&), OR(|): These are n-ary Boolean operators that return document intervals, which include hits from all/either of the sub-node queries.

Repeats Kleene operator (*+n*, *+n..m*)**:** finds sequences of hits for the same sub-query that are adjacent to each other. For example '`{Person}+2`' finds 2 adjacent person mentions; '`{Person}+1..5`' finds sequences of 1 to 5 such mentions. Unbound multiplicities are not supported, i.e. an upper limit must always be provided, which helps avoid run-away query executors.

Minus (−): The bounded negation operator finds all hits for the left hand sub-query that are not also hits for the right side sub-query. An unrestricted negation operator is not implemented, since it could match the entire index. In the case of `minus`, the maximum number of matches is restricted by the left side operand.

Sequence: this is the default operator and has no syntactic representation, as it is implied between any two queries that are not separated by any other operator. Gap markers (`[n]`, or `[n..m]`) can be included to allow a limited number of arbitrary tokens at certain locations in the sequence. For example '`{Person} [0..3] {Organization}`' would match an annotation of type Person, followed by up to three arbitrary tokens, followed by an annotation of type Organization.

2.2.3. *Structural Compound Operators*

IN: A containment operator that returns hits from the left hand side sub-node only if they are contained within hits from the right hand side sub-node. For example, the query `{Location type="city"} IN {Abstract}` would match `Location` annotations of type `city`, which occur in document abstracts (these can be either generated automatically by NLP tools, or be pre-existing structural markup).

OVER: The reverse containment operator that returns hits from the left hand side sub-node only if they are overlapping hits from the right hand side sub-node. For example, `{Reference type=publication} OVER nanomaterials` matches occurrences of `Reference` annotations with type `publication` (e.g. in the bibliography of a paper), which contain the word `nanomaterials`.

The two containment operators are useful for structural searches, such as filtering the results of another query to only the ones occurring within a document's abstract. Another example is searching for co-occurrences within the same sentence, e.g. `({Person} AND pos:verb) IN {Sentence}`.

For all compound operators, the operands can be complex queries themselves, recursively. There is no operator priority, so explicit bracketing must be used when building complex queries.

2.2.4. SPARQL-based Semantic Search Constraints

The search operators enumerated above provide the necessary support for text- and annotation-based queries. While this fully supports annotations with standard feature values, it is not sufficient for semantic annotations which have URI features.

In order to support structural search constraints against a knowledge base, Mímir has a SPARQL plug-in. It accesses a SPARQL endpoint at search time, in order to add semantic constraints to annotation queries. The plug-in is presented as a wrapper for any *annotation helper* – the module responsible for converting annotation constraints into mention IDs.

The SPARQL plug-in enables queries with semantic constraints, which are evaluated against an external knowledge repository. The dynamically retrieved knowledge then acts as an additional semantic restriction to the text- and annotation-based search.

One of the main uses of SPARQL queries is in semantics-based query expansion. This enables Mímir to retrieve documents based on information that is not present explicitly in the indexed collection. For example, a search for documents mentioning Norfolk will also retrieve documents mentioning Norwich, if SPARQL is used to query GeoNames, where the semantic relation between the two is specified.

The second main use of the SPARQL queries is for imposing semantic constraints, e.g. finding documents mentioning floods in places within 50 miles of a given location.

Section 4.2 provides a fully implemented use case, which makes extensive use of SPARQL-based query expansion, as well as of DBpedia-based semantic search constraints.

In order to explain how the SPARQL plug-in interacts with the text- and annotation-based search constraints, let us assume a document collection annotated with named entities. Each semantic annotation has an *'inst'* feature with value – the corresponding DBpedia URI. The index was then set up so that a SPARQL plug-in is configured for all annotations with *inst* features. Then the following semantic query would retrieve documents mentioning scientists born in London:

```
{Person
  sparql="SELECT DISTINCT ?inst WHERE {
    ?inst :birthPlace
      <http://dbpedia.org/resource/London>.
    ?inst a :Scientist. }"
}
```

Syntactically, this is a standard annotation query, with a single feature-based constraint. However, the SPARQL plug-in causes its execution to be different from the usual annotation queries, as follows:

1. The SPARQL plug-in is configured to intercept all annotation queries that make use of the *'sparql'* virtual constraint. We call this constraint virtual because input annotations do not actually have a feature named *'sparql'*. When the query is posed, the plug-in intercepts it, and uses the string value of the *sparql* constraint to fire a SPARQL query against the pre-configured endpoint.
2. If the SPARQL query returns a result set, then each row is parsed to extract the values bound to variables. Each `variable=value` pair is converted into a Mímir query constraint. In the example above there is only one query variable (?inst), so each result row will produce a single query constraint (e.g `'inst=<http://dbpedia.org/-resource/Rosalind_Franklin>'`).
3. For each obtained constraint, a standard annotation query is generated. All annotations are grouped into an OR compound query, which is then executed directly against the wrapped underlying annotation helper. The latter supplies the query results.

This design provides a great degree of flexibility. Arbitrary SPARQL queries can be used, and they are served by a fully-featured SPARQL endpoint. The indexed semantic annotations can have any number of URI features, which can be used in queries separately or in combination.

the configuration of the SPARQL plug-in specifies the endpoint to be used, as well as the default set of name spaces, so this information does not need to be included in each query.

2.3. Ranking of Search Results

Mímir was designed originally for information discovery searches, needed for example by intellectual property search experts (see Section 4.1) and environmental science researchers (see Section 4.2). The primary focus in this case is on enabling users to formulate very precise queries with elaborate semantic constraints. Once a matching document set is narrowed down, such users would typically investigate the entire result set. This is due to the information discovery nature of their search, which is different from seeking to retrieve a specific, best-matching document [1]. In such applications, ranking can be disabled.

In information seeking applications, however, ranking of search results is indeed necessary. Therefore, Mímir provides implementations for a set of popular ranking algorithms, and new ones can be implemented on-demand through a plug-in mechanism. For each Mímir index, a developer can choose between:

No scoring: All documents are ranked equally; results are returned in the order they are found in the index. This is the default setting, since Mímir was designed initially for information discovery applications. Later, when ranking algorithms were added, this default behaviour was preserved for backwards compatibility.

Count scoring: Matching documents are ranked according to the number of query matches contained.

BM25: Documents are ranked using the Okapi BM25 algorithm [19]. For text elements, the BM25 algorithm is applied as usual. For annotations, it is constructed from an OR between all matching annotations. For example, if a document collection contains annotations `{Person gender=[male|female]}`, and a query is presented for `{Person gender=male}`, then all occurrences of `<Person gender="male">` annotations in the document collection are treated as the same term. When queries combine both text and annotations, ranking scores are combined. For example, for the query “the name of `{Person gender=male}`” “the name of” is scored against the text index, while `{Person gender=male}` is scored against the annotation index. The BM25 scores for each part are calculated independently, and then combined as a weighted sum.

TF.IDF: Documents are ranked using the TF.IDF algorithm [20], in a manner analogous to BM25 ranking.

Hit length scoring: Documents are ranked according to the length of the matching snippets within. While the other algorithms are provided by wrapping the MG4J scorer implementations, this scoring algorithm is a stand-alone implementation, that uses the Mímir API directly. It is provided as a simple example of a scorer implementation that developers can use as a starting point for their own scoring algorithms. Its actual suitability for scoring search results may be limited.

In order to help users with *relevance judgements*, Mímir highlights matching textual snippets in each returned document. Each snippet provides the textual context around each search hit, separated by “...” if there are more than one hit per document.

3. User Interfaces for Information Seeking and Knowledge Discovery

As discussed in [3], designing easy to use and learn semantic search interfaces is both a key requirement and a major challenge.

This section discusses two kinds of semantic search interfaces. The first set of UIs are aimed at information seeking tasks, whereas the second UIs are for knowledge discovery and interactive retrieval.

3.1. User Interfaces for Information Seeking

Mímir comes with a simple web-based user interface which supports the testing and development of semantic search applications, focused on information seeking tasks. This development interface, shown in Figure 6, uses the Mímir query language which, analogous to SQL and SPARQL, is too complex to be used directly by end users. The query shown in this example is from the patent retrieval application, described in more detail in Section 4.1.

Figure 7 shows our form-based UI, which has been customised for the environmental science application[8]. This UI has a keyword search field, complemented with optional semantic search constraints, through a set of inter-dependent drop-down lists. Users can search for specific entity types (in this case Locations, Organisations, Persons, Rivers) and also specify constraints on document-level attributes. More than one semantic constraint can be added, through the plus button.

One limitation of these UIs is that they are very document-centric and do not provide details about what semantic annotations occur in the indexed document collection, (e.g. which UK counties are mentioned). In order to provide such entity-based overviews of the documents, one approach is to list all instances, for each

[8]For a different customisation of the form-based UI see http://demos.gate.ac.uk/pin/.

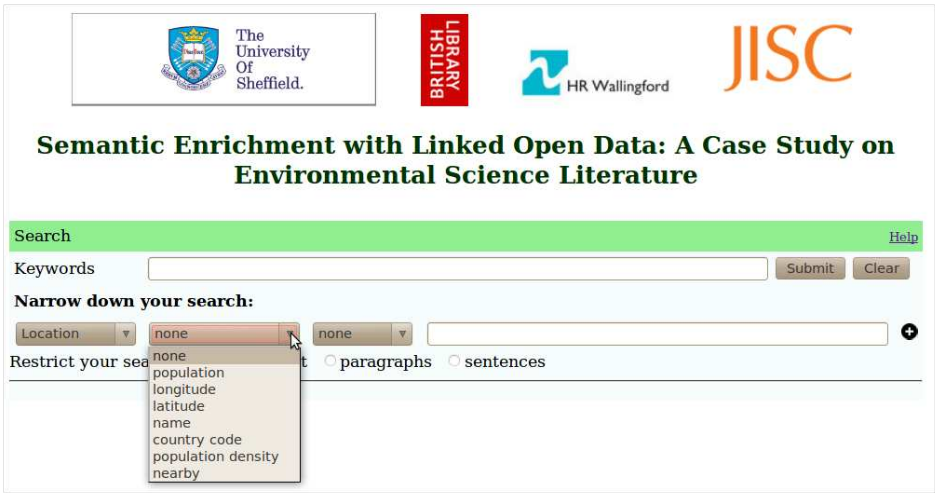
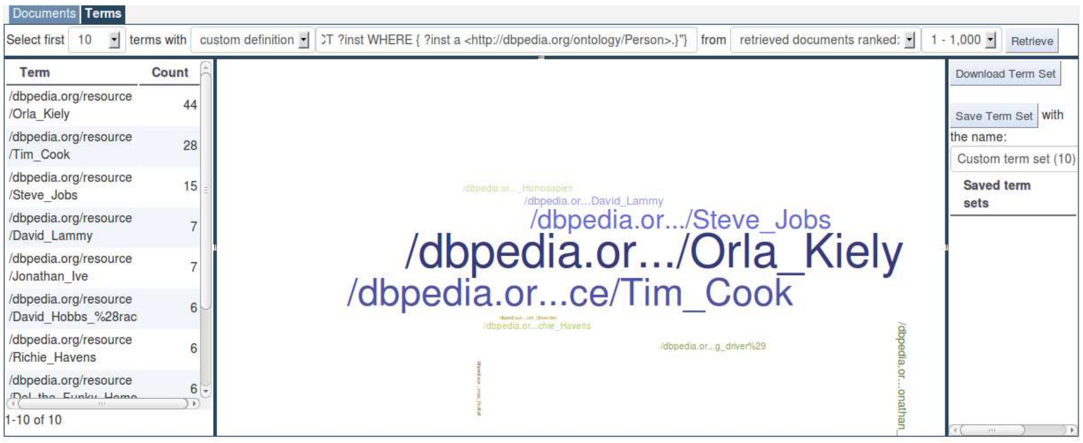

Semantic Enrichment with Linked Open Data: A Case Study on Environmental Science Literature

Figure 7: The form-based semantic search UI


Documents
Terms
Select first 10 terms with custom definition
ꓛT ?inst WHERE { ?inst a <http://dbpedia.org/ontology/Person>.}"}
from retrieved documents ranked: 1 - 1,000
Retrieve
Download Term Set
Save Term Set with the name:
Custom term set (10)
Saved term sets
/dbpedia.or..._Homosapien
/dbpedia.or...David_Lammy
/dbpedia.or.../Steve_Jobs
/dbpedia.or.../Orla_Kiely
/dbpedia.or...ce/Tim_Cook
/dbpedia.or...chie_Havens
/dbpedia.or...g_driver%29
/dbpedia.or...onathan_

Term	Count
/dbpedia.org/resource /Orla_Kiely	44
/dbpedia.org/resource /Tim_Cook	28
/dbpedia.org/resource /Steve_Jobs	15
/dbpedia.org/resource /David_Lammy	7
/dbpedia.org/resource /Jonathan_Ive	7
/dbpedia.org/resource /David_Hobbs_%28rac	6
/dbpedia.org/resource /Richie_Havens	6
/dbpedia.org/resource /Del_the_Funky_Homo	6

1-10 of 10

Figure 8: Knowledge discovery: frequently occurring terms

class, as done in KIM [2] and Broccoli [21]. This however, quickly becomes infeasible when users are searching multi-million document collections and query results have thousands of instances.

An alternative is to supplement the information seeking UI with tag clouds and other visualisations of entity co-occurrences. Mímir has recently been extended with such UIs, which we discuss next.

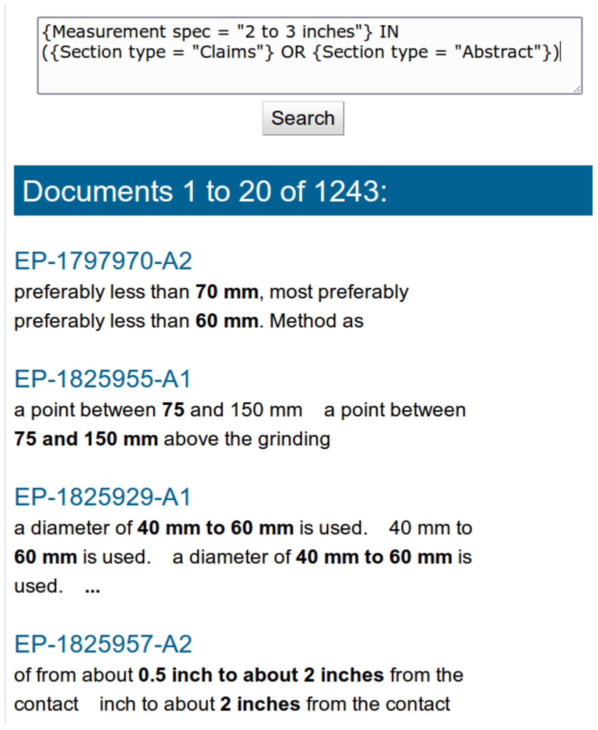
3.2. User Interfaces for Information Discovery

Information discovery tasks require more sophisticated UIs, which enable users first to narrow down the relevant set of documents through an interactive query refinement process, and then to analyse these documents in more detail, to gain useful insights. These two kinds of actions require corresponding *filtering* and *details-on-demand* information visualisations [22].

Interactive query refinement and document filtering are carried out first with an information seeking UI (e.g. the form-based UI above or the ontology-driven search UI shown in Figure 10 in Section 4.3).

Once a set of documents is retrieved, they need to be analysed in more detail. Mímir currently has two generic information discovery UIs (more will be implemented in ongoing work), which support dynamic analysis of retrieved document sets.

Figure 8 shows the general purpose Mímir UI for exploring frequently occurring terms within a dynamically created result set. Here *term* is used to refer to any word or any indexed annotation type. The results are lists of matched terms, sorted by frequency of occurrence


```
{Measurement spec = "2 to 3 inches"} IN
({Section type = "Claims"} OR {Section type = "Abstract"})
```

Documents 1 to 20 of 1243:

EP-1797970-A2
preferably less than **70 mm**, most preferably preferably less than **60 mm**. Method as

EP-1825955-A1
a point between **75** and 150 mm a point between **75 and 150 mm** above the grinding

EP-1825929-A1
a diameter of **40 mm to 60 mm** is used. 40 mm to **60 mm** is used. a diameter of **40 mm to 60 mm** is used. ...

EP-1825957-A2
of from about **0.5 inch to about 2 inches** from the contact inch to about **2 inches** from the contact

Figure 6: The developer user interface. The index being searched contains a collection of patent documents semantically annotated with relevant domain concepts. One annotation type is {`Measurement`}, used to mark expressions denoting physical measurements. The identified measurements are also normalised, which allows the correct matching of values even when the measurement unit used in the query is different from the one in the original text.

within the set of documents retrieved by the original semantic search query. Dynamically created term sets can be saved or downloaded by the user.

For this particular example, 60,000 randomly selected tweets and news pages from the UK were searched for mentions of the company Apple and then the matching documents were analysed for the 10 most frequently mentioned people within them. The term specification was in this case a SPARQL query (see Figure 8), but the UI can also show, e.g. the top most mentioned instances of a given class instead (see Figure 11 for an example of viruses).

This functionality makes use of Mímir's direct indexes which are able to find terms within a particular result set. The matched term IDs are transformed into user-friendly labels using the data stored in the annotation templates. These labels are then used to populate the term list and the term cloud.

Going one step further, the term sets constructed above can be used to find associations, as shown in Figure 9. Here two term sets are mapped to the two dimensions of a matrix, while the colour intensity of each cell conveys co-occurrence strength. The matrix can be re-ordered by clicking on any term, which sorts the axis according to the association strength with the clicked term. In Figure 9, *'Hepatitis C virus'* was first clicked, causing the X axis to sort showing which terms associate strongest with this disease. Not surprisingly, this has caused the term *'liver'* to move to the top. Next, *'liver'* was clicked, reordering the Y axis, to show on top diseases that correlate with liver.

It should be noted that this approach is data-driven and relies on counting occurrences in the dynamically retrieved document set. It can be seen, for example, that *'Hepatitis A'* appears lower on the Y axis, and the cell showing its correlation with liver is quite faint. This is caused by the fact that the initial search (for documents published in 2009, which mention viral diseases) includes only 83 documents that mention Hepatitis A, and these co-occur with mentions of the word *liver* only 5 times. The matrix would look different, if a different semantic query had been used initially, leading to a different set of matching documents used for filtering.

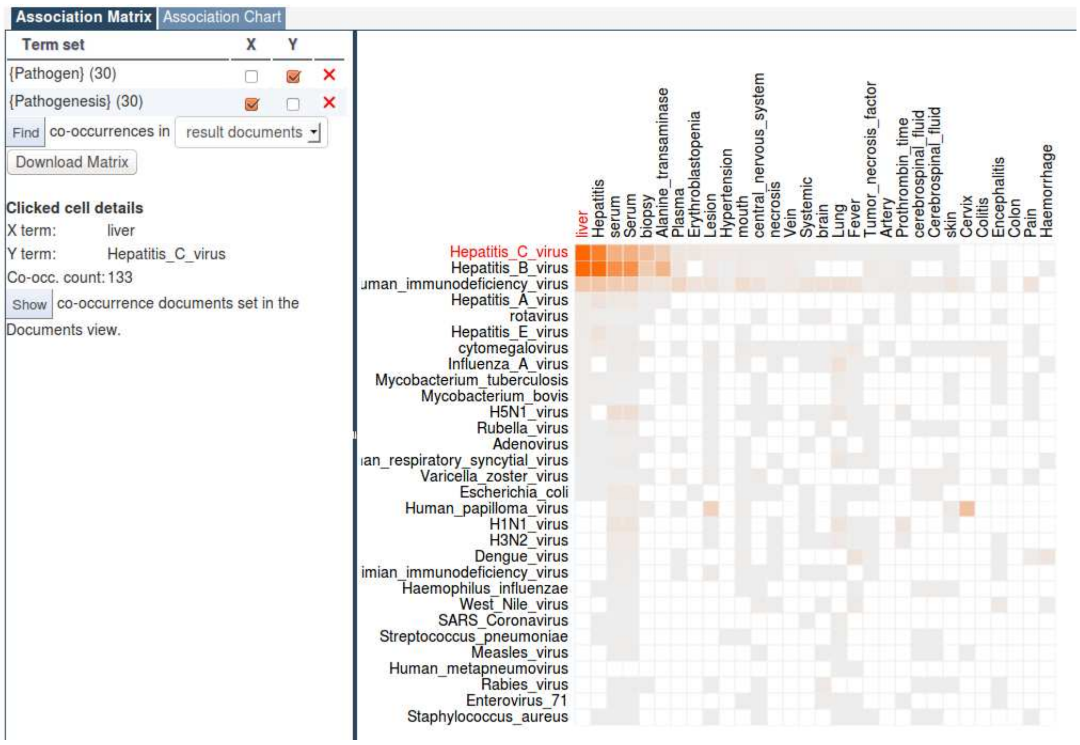
In the following section we discuss how these user interfaces were adapted to the requirements of three semantic search applications, from three different domains.

4. Three Semantic Search Applications

4.1. Prior Art and Due Diligence in Intellectual Property

When researching new product ideas or filing new patents, inventors need to retrieve all relevant pre-existing know-how and/or to exploit and enforce patents in their technological domain, i.e. they need high recall-oriented searches. The standard practice is to perform keyword searches that identify all potentially relevant documents and then analyse these exhaustively. In this set-up, ranking of results is not useful. While recall is paramount, any improvement in precision has cost saving implications.

For example, physical measurements are particularly challenging due to the many equivalent ways the same value can be expressed through using different measurement units, e.g. *inches* or *millimetres*, or different multipliers, e.g. *metres* or *millimetres*. In the case of keyword-based search, sufficient recall can only be reached if queries are made very imprecise, which leads to very large result sets and, thus, increase the human analysis time.


Association Matrix | Association Chart

Term set	X	Y
{Pathogen} (30)	☐	✓
{Pathogenesis} (30)	✓	☐

Find co-occurrences in result documents

Download Matrix

Clicked cell details

X term: liver

Y term: Hepatitis_C_virus

Co-occ. count: 133

Show co-occurrence documents set in the Documents view.

Figure 9: Knowledge discovery: term associations

In order to test the benefits of semantic search in this domain, the patent information extraction algorithms reported in [23] are used. They enrich patents automatically with linguistic and semantic annotations, mapped to a patent-specific ontology, encoded in OWL [24].

Their patent processing pipeline [23] creates annotations that fall into two broad categories: wide and deep annotation types. Wide annotations cover metadata types that apply to patents in general, irrespective of their subject area. Examples of such metadata include the identification of document sections and references to other document parts (e.g. figures, claims), or to other documents (e.g. cited literature, references to other patents). Deep annotations are specific to one or more subject areas and are of interest to specialised patent searchers. The system [23] also identifies and normalises physical measurement expressions.

For our experiments, the information extraction algorithms of [23] were used to identify measurement expressions in a collection of patents, including annotations with the normalised value of the recognised measurement. We then indexed the measurements and all other annotations with Mímir, where normalised values are stored as numeric (non-nominal) features.

Thanks to these normalised values, searching for measurement expressions becomes a standard Mímir annotation search, where constraints are expressed in relation to the normalised value. For example, one could search for 1 inch measurements using the following query: `{Measurement normalisedUnit = "m" normalisedValue = 0.0254}`. However, this is not particularly user-friendly, as it requires the user to know the exact normalised unit to use (in this case metres and not millimetres), and to perform the necessary conversions themselves. Instead, we opted for exploiting the open, extensible nature of the Mímir semantic search framework and implemented a specialised query parser plug-in to do the conversions automatically.

This works by wrapping the usual mechanism that converts an annotation query to a set of mention IDs (described in Section 2.2.1). The plug-in adds a virtual annotation feature named *'spec'*, which can be used to formulate measurement constraints in a manner familiar to the user. If present at search time, constraints using the *spec* feature are intercepted by the plug-in, parsed using the same algorithm as used for the original documents, and converted into normalised measurement values. These are then used to formulate an underlying Mímir query that uses the actual annotation features. An example of this is shown in Figure 6: the user searches for *"2 to 3 inches"* and the system finds documents that contain the expressions *'60mm'* and *'2 inches'*. The user-supplied query was converted transparently into a query similar to:

```
{Measurement normalisedUnit ="m"
  normalisedValue >= 0.0508
  normalisedValue <= 0.0762}
```

The actual query is more complex, since it also needs to match measurement ranges in the original document (e.g. *between '75 and 150 mm'*), for which not both of the end-values fall within the query interval. As can be seen in Figure 6, *'75mm'* and *'75-150mm'* were matched, while *'150mm'* was not.

Searching for measurements in this way arguably benefits from the semantic interpretation of the original text content, since the search is not based on the actual words. For example, the query *'2 to 3 inches'* and the result *'60 mm'* have no textual overlap at all. However, due to special requirements, it does not do so through the use of formal semantics and ontologies.

The next application demonstrates use of ontologies and Linked Open Data to augment semantic search.

4.2. Environmental Linked Data

Environmental science is a broad, interdisciplinary subject area where information discovery and management is often a challenge [25].

Linked Open Data (LOD) offers an opportunity to improve the process of information discovery and sharing through unique, machine-readable, interlinked open vocabularies. For example, a user searching for flooding in Britain would be able to find a report with a chapter on water levels at the Thames barrier, thanks to the knowledge in Geonames which states that the Thames barrier is located in England.

In other words, by exploiting additional semantic knowledge from LOD resources, users were able to benefit from Mímir's semantic search and find reports that would not have been picked up by full-text search alone (see Section 5.3 on the user-based evaluation).

The first step in this joint project with the British Library was to apply LOD-based semantic annotation. We enriched 10,000 environmental science documents and associated metadata with term and entity URIs from DBpedia [12] and GeoNames, as well as with linguistic information, such as part of speech.

The resulting documents, annotations, and URIs were indexed in Mímir and it was configured to execute SPARQL queries against two semantic repositories (one for GeoNames and one for DBpedia). GeoNames was used as a source of rich knowledge about locations (e.g. NUTS administrative regions, latitude, longitude, parent country, population count), while DBpedia provided knowledge about people, organisations, and products.

The main challenge was to customise the form-based semantic search user interface (Figure 7), so as to hide the complexities of the Mímir query language (SPARQL in particular), while allowing users to issue powerful semantic search queries.

A specific requirement was support for location-based queries, so users can narrow down results by name, geographic coordinates, population, population density and country code. Users also wished to search for locations that a river flows through (see the corresponding Mímir semantic query below), as well as documents mentioning locations near a given location.

```
{Mention dbpediaSparql =
  "SELECT DISTINCT ?inst
     WHERE {
       ?inst a dbont:River .
       ?inst dbprop:city ?x .
       FILTER (REGEX(STR(?x),
         \"Gloucester\", \"i\")) }"
}
```

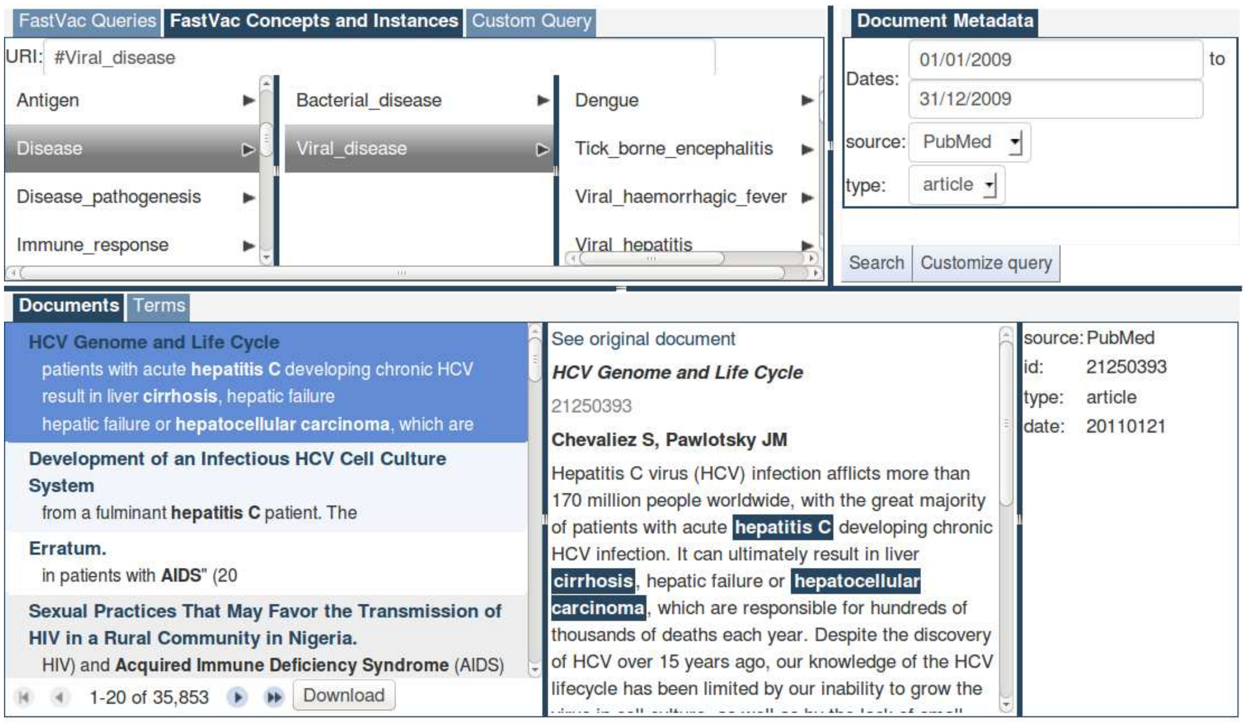
For string-value properties, if `is` is chosen from the third list in Figure 7 instead of none, then the value must be exactly as specified (e.g. Oxford), whereas `contains` triggers sub-string matching, (e.g. Oxfordshire is matched as a location name containing Oxford). In this way, a user searching for documents mentioning locations with name containing Oxford, will be shown not only documents mentioning Oxford explicitly, but also documents mentioning Oxfordshire and other locations in Oxfordshire (e.g. Wytham Woods, Banbury). In the latter case, knowledge from DBpedia and GeoNames is used to identify locations in Oxfordshire.

4.3. Immunology Literature Review

In this project, the end users are immunology experts performing systematic literature reviews. The target use case is speeding up new vaccine development through helping experts identify links and correlations between domain concepts and entities that occur in previously published research literature.

With this aim, a prototype was built that uses GATE to annotate semantically a large document collection (9.5 million documents), which comprises abstracts of all potentially relevant papers from the PubMed library[9] and the Cochrane Collaboration[10].

[9] `http://www.ncbi.nlm.nih.gov/pubmed`
[10] `http://www.cochrane.org/`



Dates: 01/01/2009 to 31/12/2009
source: PubMed
type: article

HCV Genome and Life Cycle
patients with acute **hepatitis C** developing chronic HCV
result in liver **cirrhosis**, hepatic failure
hepatic failure or **hepatocellular carcinoma**, which are

Development of an Infectious HCV Cell Culture System
from a fulminant **hepatitis C** patient. The

Erratum.
in patients with **AIDS**" (20

Sexual Practices That May Favor the Transmission of HIV in a Rural Community in Nigeria.
HIV) and **Acquired Immune Deficiency Syndrome** (AIDS)

1-20 of 35,853

HCV Genome and Life Cycle
21250393
Chevaliez S, Pawlotsky JM
Hepatitis C virus (HCV) infection afflicts more than 170 million people worldwide, with the great majority of patients with acute hepatitis C developing chronic HCV infection. It can ultimately result in liver cirrhosis, hepatic failure or hepatocellular carcinoma, which are responsible for hundreds of thousands of deaths each year. Despite the discovery of HCV over 15 years ago, our knowledge of the HCV lifecycle has been limited by our inability to grow the

source:	PubMed
id:	21250393
type:	article
date:	20110121

Figure 10: The immunology document search interface. The top panels are used to formulate the search constraints. On the left, the principal query is defined by picking an ontology class or instance; on the right additional constraints can be entered, based on document metadata.
The lower panels show (left to right): the list of matched documents, the content of the current document, and some additional document metadata.

A domain ontology was built with the users, covering concepts and properties relevant to immunology and vaccine development. The same ontology is used by the semantic annotation pipeline and Mímir.

The annotated documents are then indexed with Mímir, and the resulting index is presented to the users through a customised version of Mímir's information discovery user interfaces.

Figure 10 shows the semantic knowledge discovery tool, running on the collection of over 9.5 million documents. Users formulate queries by selecting values from a set of column browsers, drop-down lists and date pickers; no actual query has to be entered manually.

In this example, a user has requested documents mentioning *viral diseases*, published in PubMed in 2009. As a result, the system has generated the following query:

```
{Disease sparql = "SELECT ?inst WHERE {
  ?inst a :Viral_disease}"}) IN
{Document source = "PubMed"
  type = "article"
  date >= 20090101 date <= 20091231}
```

As can be seen, the use of semantics in the query has allowed to match mentions of various types of hepatitis and AIDS with the *'viral disease'* query.

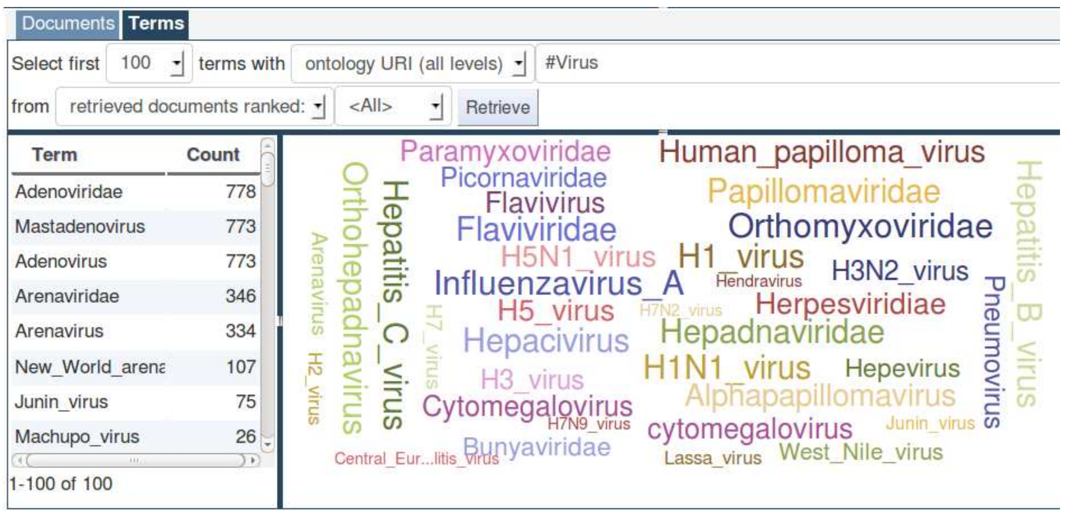
Once users have refined their information needs through interactive semantic searches, they next use the knowledge discovery UIs to get details on frequently occurring terms (Figure 11) and important co-occurrences (Figure 9).

In more detail, Figure 11 shows Mímir's term frequency UI (Figure 8), this time running over the immunology index. In this particular example, the user has selected the ontology concept `Virus` which matches all semantic annotations associated with that class. Instead of showing the raw URIs, the UI has been configured to show RDF labels instead, for user friendliness.

5. Evaluation

In this section we look at space cost for building indexes, and at time efficiency for performing different types of searches. Lastly, the results of a users-based evaluation experiment are presented.

The experimental server had 24 cores (two AMD Opteron 2.3GHz processors with 12 cores each) and 64GB RAM, using MG4J version 5.2.1.



Select first 100 terms with ontology URI (all levels) #Virus

from retrieved documents ranked: <All>

Term	Count
Adenoviridae	778
Mastadenovirus	773
Adenovirus	773
Arenaviridae	346
Arenavirus	334
New_World_arena	107
Junin_virus	75
Machupo_virus	26

1-100 of 100

Figure 11: Knowledge discovery: frequently occurring terms

The experimental setup for Section 5.1 was deliberately single-threaded, opening the index files directly using the lowest level Mímir API calls. In this way we could ensure exact timing of only the search process, isolated from any overheads introduced by the web application. None of the experimental queries in Section 5.1 involve any SPARQL.

The evaluation of complex semantic queries in Section 5.2 used the live index instead of the low-level API. The SPARQL queries were executed against Sesame (version 2.3.3), running on the same server. The ontology was in ”Sesame native” format.

5.1. *Indexing and Search Efficiency*

Indexing efficiency is evaluated using the index from the immunology application described above. Index sizes are reported in Table 1. When compared to the Wikipedia dumps used for evaluation of other semantic search systems (e.g. [21]), our dataset is 3 times larger in terms of number of tokens and 1.4 times larger in terms of raw storage space.

As can be seen, the immunology index includes just over 9.5 million documents (about 46 GB of uncompressed text). The index configuration includes 36 annotation types, as well as 3 token features (string, morphological root and part-of-speech). The document collection included over 743 million individual annotation occurrences, and 3.4 billion tokens.

The resulting annotation indexes and templates take up a total of 2.3GB of disk space, while the largest token index takes up 10GB. It can also be seen that, although all token indexes include the exact same number of occurrences, their size differs according to how dense they are. The part-of-speech index is the most dense (i.e. it has the smallest number of terms, with longer average posting lists) and it takes up about one third of the space required by the string and root indexes. The fact that the root and string indexes are of similar size is not surprising, since most English words are not inflected, so normalising words to their morphological roots does not reduce information content significantly.

Annotation occurrences require, on average, similar storage space to token strings. Conceptually, an annotation contains more information than a word, so this result confirms our hypothesis that using annotation template IDs results in very compact indexes. The only penalty incurred in supporting level-1 and level-2 IDs is the additional disk space for the template data, which at 9.3MB is negligible. The resulting reduction in annotation index sizes also benefits the execution speed, since a smaller index is also faster to search.

Exactly how these index sizes relate to the data size, and thus the rate at which the index would grow as more documents are added, is a complex question. There are many different factors to consider, but all should be subject to linear upper bounds. For the token indexes the document content store and the inverted index posting lists will grow linearly with the total number of *token occurrences* in the index. The direct index will grow linearly with the smaller total number of *distinct tokens in each document* (because the direct index does not store the positions at which a term occurs, only the number

Document Collection	
Documents count	9, 551, 404
Annotation occurrences	743, 163, 979
Token occurrences	3, 400, 912, 569
Plain text size	46.28 GB

Index sizes	
Annotation templates	9.3 MB
Annotation indexes	2.29 GB
Document content and metadata	17.3 GB
Token indexes	23 GB
part-of-speech index	(3.27 GB)
morphological root index	(9.73 GB)
string index	(10 GB)
Total Index Size	**42.6 GB**

Table 1: Statistics for the immunology index. N.B. all multiples use powers of 10, so the units are e.g. *GB* and not *GiB*.
All indexes were configured to include both inverted and direct indexes. If only inverted indexes were used, the sizes would be 30..40% smaller.
Document content and metadata includes the original document text and stored metadata that is presented in the search results.

of times it occurs within each document). MG4J uses various encoding schemes to store numeric data such as term IDs and positions in as few bits as possible while still maintaining good decoding performance, so in practice the constant factors in these linear bounds will be small, and depend on index density.

In order to evaluate the search-time execution speed of Mímir, we used the same immunology index described above. It is to be expected that the time taken to answer a given query depends both on the complexity of the query and the total number of result documents that are returned. To measure these effects, we started by profiling the execution time of each of the different query operators with regard to the number of results generated.

For each operator we identified a query that was guaranteed to generate a large number of results. In each case we ran the same query 20 times and selected the best execution time. We also applied a delay of 1 second between executions, to force the Java JIT compiler to *forget* whatever optimisations it had applied to the tight loops. Without this delay, subsequent execution times can be up to 3 times shorter, but this would not be a natural use case, as the exact same query is not usually executed repeatedly. Applying the delay has led to a balanced set of values, eliminating the bias for small values on the first measurement, and approximating a Gaussian distribution.

For each query, execution times were then plotted against the number of returned documents. The measured times include enumerating all returned document IDs, but do not include the production of snippets. Snippets, and other data needed to present the results to the user would normally only be required for the first page of results, typically the first 10 – 100 documents. Given a fixed number of documents per page, the construction of such results page is performed in constant time.

The queries we used are:

- *‘a’*: the word *‘a’* is likely to occur in most documents.
- *‘{Document}’*: each document in the collection was annotated with a `{Document}` annotation, so this annotation query is guaranteed to match all indexed documents.
- *‘a AND a’*, *‘a OR a’*: simple Boolean queries that will match the same document set as *‘a’* above.
- *‘{Document} IN {Document}’*: containment query that matches all documents as all `{Document}` annotations are co-extensive with a `{Document}` annotation. This is an abuse of the semantics of the containment operator, but it allows us to construct a query that will find results in all documents. The the opposite containment operator *‘OVER’* would perform the exact same logic so we have not measured it separately.
- *‘(a OR the) MINUS the’*: an artificial query that will match most documents whilst using the *MINUS* operator. It finds all occurrences of either *‘a’* or t̀he’ which are not *‘the’*. This is, of course, not an optimal way to search for occurrences of *‘a’*.
- *‘pos:DT pos:JJ pos:NN’*: finds sequences of words with parts of speech *determiner*, followed by an *adjective*, and a *noun*. Effectively, this matches noun phrases, which are very frequent in English, thus ensuring a large number of results.

It should be noted that the `Term` and `Annotation` queries are essentially wrappers around the MG4J `Index Iterator` operator. All the others are implemented by Mímir, even in cases where MG4J had an equivalent operator (e.g. `AND`). This was done to allow direct access to the compound Mímir index, and to reduce the number of format conversions that needed to be performed between the MG4J representation of a result set and the one used by Mímir.

The results are presented in Figure 12. It can be observed that all operators behave linearly with regard to the number of documents retrieved. As shown in the left side plot, base operators (string and annotation), as well

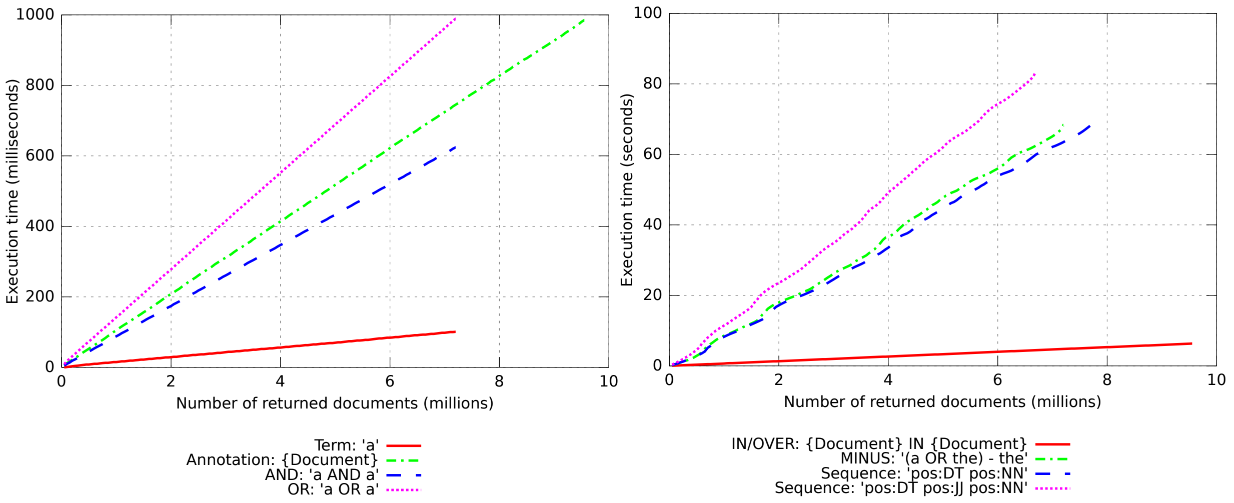


Figure 12: The execution times for different types of query. Each unit on the X axis represents a set of one million documents returned as a result of each query. The two different graphs use different scales for their respective Y axes: the one on the left goes up to 1 second, while the one on the right covers more complex queries and has a maximum value of 100 seconds. Shorter lines correspond to queries that do not return the maximum number of results (9.55 millions), which can happen when a query does not find hits in every document in the index.

as Boolean ones, have response times of a few hundred milliseconds, each being able to match seven million or more documents in under one second.

The right side graph shows the statistics for the more complex operators. The containment operators, that are used for performing structural searches over document annotations, are the most efficient with an execution time of about 6 seconds. In this particular case, both the left and right side operators are the annotation query `{Document}`, which was separately measured at about 1 second, so the time used by the actual containment operator is about 4 seconds (i.e. $6 - 1 - 1$). The `MINUS` and sequence operators are significantly slower, taking between 54 and 74 seconds to match six million documents.

As expected, the execution time for the sequence operator increases in line with the length of the sequence: in order to match $n + 1$-length sequences, the system has to find all n-length ones and try to expand them. To investigate this further, we also tested sequence queries of 4 and 5 terms. Because these queries are more constrained, they match far fewer documents. In order to get comparable results we measured the time taken to retrieve the first $100,000$ documents in each case. As shown in Figure 13, the execution time increases with the length of the sequence in a higher than linear fashion. This suggests that sequence queries longer than 3 terms should be used with caution, and avoided wherever possible. However, in practice, users tend to for-

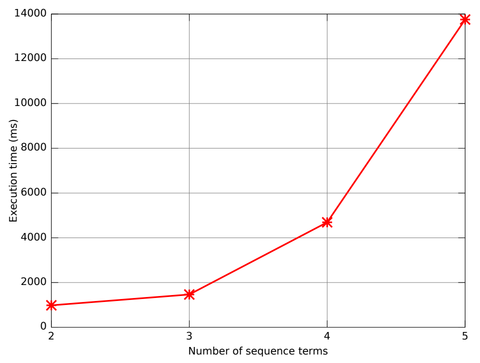


Figure 13: The execution times for retrieving the first 100,000 documents for sequence queries of different lengths.

mulate search queries of (on average) around 2.4 terms [26], which indicates that in most cases there will be no significant delay.

The reason for this higher than linear increase in execution time is due to the way Mímir implements the sequence operator:

1. First it finds all documents that contain all the terms required for the sequence (essentially an AND query), disregarding where in the documents they occur;
2. For each such document, it retrieves the positions of each of the query terms, and return as results

only occurrences where the terms occur in the correct sequence.

As the length `k` of the sequence query grows, the execution time of the first step increases slower than linearly. In the second step, for every document found in step 1, there are `k` sorted lists of positions within the document. Then calculation of all valid sequences is carried out, which is a complex operation resulting in higher than linear increase in query execution time.

5.2. *Evaluating a Complex Semantic Search Query*

The queries above were chosen specifically to match as large a fraction of the entire index as possible, in order to allow us to profile search efficiency. However, they are not representative of actual semantic search queries formulated in practical applications.

Next we look in some detail at an actual user-generated query. This was produced by an immunology domain expert, who has been trained to use Mímir, and has become a proficient user of the query language. The expert was presented with the following information need:

> For the relevant pathogens, what antigens are protective or associated with disease amelioration or control of infection, in natural or experimental infection in Man?

Based on that request, they produced the Mímir query shown in figure 14. When executed against the full index, this query returns 899 documents, and takes 35 seconds to execute. In practice, users did not perceive this response time as problematic, due to the information discovery nature of their searches.

We look next at the main constituent parts of this query, their intent, and their execution times.

The first query segment (lines 3-5, inclusive) finds pathogens, described as either annotations of type `{Pathogen}` or of type `{Disease}`, as long as they refer to bacterial or viral diseases. In this case, the expert was not concerned with diseases caused by parasites. This query on its own matches $2,375,025$ documents in just under one second.

The next segment (lines 9-17, inclusive) finds relationships of type protection or control if they overlap both a mention of a *specific antigen* and an *immune response*. Furthermore, the results should only include occurrences inside sentences that express an *assertion*[11]. This query segment matches $67,769$ documents, in 33 seconds.

```
1  (
2    (
3      {Pathogen} OR
4      {Disease sparql = "SELECT DISTINCT ?inst
       ↪  WHERE {?inst a :Bacterial_disease}"} OR
5      {Disease sparql = "SELECT DISTINCT ?inst
       ↪  WHERE {?inst a :Viral_disease}"}
6    )
7    AND
8    (
9      (
10       (
11         {ProtectionAssociation} OR
12         {ControlAssociation}
13       )
14       IN
15       {Sentence type = assertion}
16     ) OVER
17     ( {SpecificAntigen} AND {ImmuneResponse} )
18   )
19   AND
20   (
21     {AnimalsAndModels sparql=
22         "SELECT DISTINCT ?inst WHERE { {?inst a
         ↪  :Man} UNION {?inst a :Human_study}
         ↪  }"}
23     IN
24     {Sentence type="assertion"}
25   )
26 )
27 IN
28 (
29   {Document}
30   MINUS
31   (
32     {Document}
33     OVER
34     (
35       root:vaccinate OR root:immunize OR root:
         ↪  immunise OR immunogen OR
36       vaccinogen OR {Vaccine} OR
37       {ImmuneResponse class = ":
         ↪  Artificial_active_immunity"}
38     )
39   )
40 )
```

Figure 14: Mímir query corresponding to a more complex semantic search

The execution time for this segment is quite high, so we decomposed it further. It turns out that most of the time is taken by the `IN {Sentence type = assertion}` operation. It takes 14 seconds to enumerate all assertion sentences in the entire collection, since almost all documents contain some. This creates a very large result set on the right side of the `IN` operator, which causes it to slow down.

As an experiment, we ran the same query segment without the `IN` operation and the execution time was reduced to 7.5 seconds. However, the number of documents matched has now increased to $73,916$, which shows that the constraint we removed was actually necessary.

The query segment at lines 21-22 is looking for mentions of the human species or human studies. It matches

[11]as opposed to a *question*, or a *negation*

$1,102,517$ documents in 0.2 seconds. As above, adding the further constraint at lines 23-24 slows it down significantly to 16 seconds, but reduces the number of matching documents to $994,326$.

Finally, the query segment at lines 29-39 finds all the documents that **do not** contain mentions of vaccine-induced infection or immunity. It must be noted that the original query asked for *natural* or *experimental* infections, not the ones caused by vaccination. The expert is using the MINUS operator to exclude all documents that mention vaccination or related terms. This segment matches $8,724,946$ documents in just under 9 seconds.

The execution time for the full query is significantly shorter than the sum of its constituents (35 seconds compared to a sum of $1 + 33 + 16 + 9 = 59$ seconds) because when executed together the various operators reduce each other's search space.

A less rigorous but practically important efficiency measurement is the time required to calculate the association matrices, such as the one shown in Figure 9. To populate each cell of this matrix, the user interface fires an AND query, in which both terms are annotation queries with semantic constraints. This particular example includes 900 cells and was populated in under 3 seconds. The user interface allows the construction of matrices of up to $100x100$ cells, which are typically visualised in less than 9 seconds.

5.3. User-based Evaluation

In order to evaluate the usability and learning overhead of the form-based Mímir UI, a small-scale user evaluation was conducted during a workshop, hosted by the British Library. 23 participants attended the workshop, who could be broadly classified into environmental scientists (43%), users interested in applications of semantic search to other domains (22%) and semantic technology developers (35%).

Users were asked to compare the results from keyword-based search against those produced by the Mímir semantic search. All participants were asked to complete four search tasks:

Task 1. Find documents on flooding on rivers flowing through Gloucester

Task 2. Find documents on flooding in places near Sheffield

Task 3. Find documents on flood risk management in locations with population less than 15000 inhabitants

Task 4. Find the areas at risk of surface water flooding in London

	Task 1	Task 2	Task 3	Task 4
Task completion rate	100%	88.24%	88.24%	76.47%
Keywords	47.06%	70.59%	35.71%	69.23%
Sem. search	82.35%	70.59%	96.43%	73.08%

Table 2: User Evaluation Results

Participants first completed each task using keyword search only, and then re-formulated the query to include also semantic search constraints from the form-based semantic search interface. For each task, participants were asked to write down the queries they used.

More formally, the experiment has a repeated measures, task-based design, i.e., the same participants interacted with the two versions of the system, in order to complete a given set of tasks. Prior to the experiment, participants were shown a 10 minute demonstration of the semantic search interface. Afterwards, participants were given 30 minutes to complete the four tasks, using the two search methods.

As can be seen from Table 2, task completion rates vary. This is mainly due to the higher complexity of tasks 2, 3, and 4, but also several users did not attempt the later tasks, because they ran out of time. Nevertheless, each task was completed by at least 13 participants.

The percentage of participants who found relevant documents using keyword search only (second row in Table 2) also differs between tasks. The success rates for tasks 1 and 3 are low, due to needing knowledge which is not present explicitly in the documents. Namely, in task 1 this is about which rivers flow through Gloucester, and in task 3 – which places in the UK have population less than 15,000 inhabitants. Task 4 is about searching for risk areas in London, where again some relevant documents do not mention London explicitly.

Thirdly, participants stated that results obtained through Mímir search were better than those from keyword search alone. However, as task success rates indicate, not all users were able to use the semantic search interface effectively.

Lastly, we also evaluated the usability of the semantic search UI, through a user questionnaire, which was returned by 16 workshop participants (after they completed the search tasks). We used the SUS usability questionnaire [27], with questions on usage and complexity of the UI, and its ease of learning.

The mean questionnaire score is 72.3, which indicates that the form-based semantic search UI has good overall usability (a good SUS score needs to be over 68). Standard deviation is 10.2. 69% of participants scored the system above 68 overall.

In more detail, 9 of the 16 (56.3%) participants agreed or strongly agreed that they would use the form-based semantic search UI frequently. Another 6 participants were neutral and only 1 participant strongly disagreed. 14 of the participants (87.5%) disagreed or strongly disagreed with the statement that the semantic search UI is unnecessarily complex and 2 were neutral. 15 of the 16 participants (93.75%) felt they can use the system without needing to learn more about it first.

Overall, from the questionnaire we can conclude that there are no major issues with the form-based semantic search UI, which make it complex or hard to use for the majority of users. Almost all users learnt to use integrated semantic search successfully after a short demonstration. However, an open question remains as to how easy would users find it without any prior demonstration. This we plan to address in future work.

6. Related Work

Since Mímir is concerned with searching both full-text and document structure, research on XML search is broadly relevant (see [28] for a survey), including query languages, such as TeXQuery [29], which combine full-text predicates and an XML query language such as XQuery[30]. Since linguistic annotations could be exported as XML markup on documents, it is also possible to use such approaches for querying linguistic annotations, in addition to full text and document structure. Similarly, research on keyword search in databases (see [31] for a survey) is also broadly relevant. However, semantic full-text search approaches, such as Mímir and other mentioned below, go one step further, by making use of formal semantics and logical inference, in order to interpret semantic annotations and queries by tapping into knowledge external to the documents (e.g. DBpedia and other LOD resources).

Semantic search is a term that has been used to describe a wide range of search approaches, that go beyond simple keyword-based search [32]. This section discusses the most relevant, recent semantic search systems of each kind.

Concept search (e.g. [33]) is a modification of traditional full-text search, where concepts instead of words are indexed and searched. Another example is the ontology-based IR model proposed in [34], where semantic entities are indexed in the documents where they appear. Search is carried out via a natural language query, which is transformed into SPARQL to retrieve all matching entities from a given knowledge base, followed by retrieval and ranking of documents containing these entities. These approaches differ from Mímir's integrated semantic search since they ignore document structure and use text-only queries, which are analysed to derive the corresponding concepts.

With respect to searching linguistic annotations, dedicated corpus search approaches have been developed [35, 36], some of them inspired by research on XML search [28]. For example, [36] propose a modification of XPath, in order to account for the horizontal, in addition to the vertical structure present in linguistic annotations. Fundamentally, only the structured linguistic data is searched, which falls short of integrated semantic search, which also includes full-text content and external knowledge sources.

Our semantic search approach also differs from semantic web search engines, such as Swoogle [37] or SemSearch [38], in its support for implicit semantics, discovered through NLP techniques, rather than reliance on explicitly encoded semantics in an ontology or RDFa annotated documents. Similarly, semantic-based facet search and browse interfaces, such as /facet [39], tend to show the ontologies explicitly, whereas annotation-based facet interfaces (see KIM below) tend to hide the ontology and instead resemble more closely "traditional" string-based faceted search.

Much of the semantic search work has focused specifically on searching knowledge graphs, typically encoded in RDF [40] or OWL [24], and stored in a database or a semantic repository. This problem has been referred to as *ad-hoc object retrieval* [7] or *entity(-oriented) search* [41, 42]. Methods include graph traversal [43] and reasoning [44]. In this case, the search need is formulated in a structured query language, such as SPARQL [4]. There has also been work on mapping keyword queries to semantic queries over RDF data, e.g. [42, 7] for entity search and the QUICK system [45] for building semantic search queries from keywords. This kind of search could be referred to as *structured semantic search*, because it operates on structured semantic knowledge, coupled with inference techniques. It is particularly suited to answering instance-type queries, such as "Which UK towns are within 100 miles of Sheffield".

The semantic search systems most similar to Mímir are the *semantic full-text search* or *hybrid search* approaches. The KIM (Knowledge and Information Management) platform [46, 2], was among the first such systems to implement semantic search, both over RDF knowledge bases via SeRQL (later SPARQL) and over semantically annotated document content. KIM uses hybrid queries mixing keywords and semantic restrictions and comes with a number of user interfaces for semantic search and browsing. Mímir differs from KIM in supporting structural compound operators, which enable semantic search also over linguistic annotations and document structure markup.

GoNTogle [47] is a search system that provides keyword, semantic and hybrid search over semantically annotated documents. The semantic search replaces keywords with ontological classes. Results are obtained based on occurrences of the ontological classes from the query within the annotations associated with a document. Finally, the *hybrid search*, comprises a standard Boolean AND or OR operation between the result sets produces by a keyword search and a semantic search. The only type of annotation supported is associating an ontology class with a document segment. Another similar system is Semplore[48] which uses conjunctive hybrid query graphs, similar to SPARQL, but enhanced with a "virtual" concept called keyword concept W. However, both GoNTogle and Semplore do not have support for structural searches of any kind, nor for other types of linguistic information.

Lastly, the Broccoli system [21] provides a user interface for building queries, combining text-based and semantic constraints (encoded as entity mentions in the input text, with URIs). The association between text and semantics is encoded by means of the *occurs-with* relation which is implied whenever mentions of words and ontological entities occur within the same *context*. The *contexts* are automatically extracted at indexing time, and rely mainly on syntactic analysis of the document sentences and extraction of syntactic dependency relations. The *occurs-with* relation is essentially a representation of the underlying phrase structure of the input document. This works well, and adds weight to the argument that structure is important in search.

One key difference between Broccoli and Mímir is that, by default, Broccoli performs entity-oriented search (although document search is also supported), whereas Mímir returns documents.

Another difference is in the way document structure is modelled. Broccoli is only concerned with co-occurrence of words or entity mentions within the same context. It chooses a context definition that is aimed at best representing relatedness. Having defined what contexts are, it then uses that decision to influence the design of the indexing system, in a way that maximises performance.

By contrast, Mímir makes no assumptions about what type of document structure will be most relevant to a given semantic search application. It simply provides a model for *annotations* over the input text, which represent arbitrary metadata associated with arbitrary text extents inside the input documents. These annotations can be used to represent document structure (e.g. sentences, sections, bibliography), entity mentions with links to an ontology, linguistic information (e.g. noun phrases), or simple plain 'tags' that are relevant to the user's domain (e.g. *ProtectionAssociation*). Mímir then provides a generic implementation for indexing and interrogating these annotations along with the textual content. This makes Mímir a generic tool, that can be used with data from any domain, as demonstrated by the quite different use cases presented in Section 4. For any new application, we require a representation of the target domain (e.g. an ontology), and a text analysis pipeline that can annotate input text to identify mentions of relevant entities and document structure. Given these, Mímir can be used to power domain-relevant searches over any document collection that was processed with the provided analysis pipeline.

Unsurprisingly, this added generality comes with added costs, the first victim being user-friendliness. Because of its more constrained task definition, Broccoli can provide dedicated User Interfaces (UIs) that make the interaction with the system simpler and more intuitive. The UIs also provide context-sensitive suggestions, while users type, e.g. class or relation names, to lower the cognitive load and improve usability.

For Mímir, the query language is complex, combining the system's own idiosyncrasies with the relatively high complexity of SPARQL. The user is also required to have a good familiarity with the domain ontology and the annotation schema. This complexity can be mitigated through the development of reusable user interfaces, some examples of which are shown in Section 4. However, these usually require some customisation effort to make them optimal for a given application and domain.

Broccoli also takes advantage of its fixed definition for contexts when designing the index structure. Its indexes are optimised to represent co-occurrence of terms (either words or entities) within the same context. This gives it a very good performance profile when answering queries. One could conceive of an annotation schema that could be used with Mímir to represent that same *occurs-with* relation that Broccoli uses. For example, one could use the Broccoli text processing pipeline to identify entities, and also create `{Context}` annotations, which are then all indexed in Mímir. The *occurs-with* relation could then be interrogated using a query like `(#item1 AND #item2) IN {Context}`, where the two items can be terms or annotations representing entities. This will allow the formulation of queries equivalent to the ones answered by Broccoli, and which will return the same results. However, the execution time will be significantly slower – in the order of seconds for Mímir, compared to hundreds of milliseconds for Broccoli.

Another difference comes from the way Mímir integrates SPARQL-based semantic constraints. Because the SPARQL side of the query is executed externally, in isolation, it cannot be filtered based on non-SPARQL constraints. For example, one cannot search for "mentions of a `{Person}` born in a `{City}` that occurs in the same document as the word *smog*". This would require the list of cities that co-occur with *smog* to be fed into the SPARQL query, and this is currently not supported by Mímir, although we plan to add it in future work.

To summarise, the novelty of the Mímir semantic search framework lies in its tailoring to serendipitous information discovery tasks, supported through a number of visualisations, including co-occurrence matrices and term clouds, as well as an interactive retrieval interface, where users can save, refine, and analyse the results of a semantic search over time. The more well-studied information seeking searches are also supported, including ranking of search results. To the best of our knowledge, the Mímir framework is the first open-source semantic search platform of this kind.

7. Conclusion

This paper introduced the Mímir semantic search framework, which can index and search full-text content, document structure, linguistic annotations, and ontologies. This kind of semantic search is particularly beneficial where search results require knowledge not contained explicitly in the document content (e.g. documents about flooding in UK cities with population under 50,000 people).

Mímir combines conventional full text Boolean retrieval, structural annotation graph search, and SPARQL-based concept search. A major advantage is Mímir's extensible architecture, where new methods for indexing, result ranking and query interpretation can easily be added through pre-defined APIs.

Future work on Mímir will address utilising relevance feedback, to train the retrieval algorithms underneath. Similarly, user adoption could be improved further through a "more like this" functionality, to help users with refining semantic search queries.

There is also ongoing work on extending the Mímir framework towards indexing and search over content streams, moving away from the classic offline indexing and online search architecture.

Lastly, with respect to improving the usability of the Mímir query language, we are planning to investigate the use of a natural language query interface. This would build on our earlier work on natural language interfaces to ontologies [49].

Acknowledgements

This work was supported by the EU FP7-ICT Programme under grant agreements No. 296322 AnnoMarket and no.287863 TrendMiner, and a UK EPSRC grant No. EP/I004327/1. We also wish to thank the editors and reviewers for their extensive and constructive feedback, which helped us improve the paper significantly.

References

[1] P. Pirolli, Powers of 10: Modeling complex information-seeking systems at multiple scales, IEEE Computer 42 (3) (2009) 33–40.

[2] A. Kiryakov, B. Popov, D. Ognyanoff, D. Manov, A. Kirilov, M. Goranov, Semantic annotation, indexing and retrieval, Journal of Web Semantics 1 (2) (2004) 671–680.

[3] H. Bast, F. Bäurle, B. Buchhold, E. Haussmann, A case for semantic full-text search, in: Proceedings of the 1st Joint International Workshop on Entity-Oriented and Semantic Search, JIWES '12, ACM, 2012, pp. 4:1–4:3.

[4] E. Prud'hommeaux, A. Seaborne, SPARQL Query Language for RDF, W3C recommendation – 15 january 2008, W3C, http://www.w3.org/TR/rdf-sparql-query/ (2008).

[5] K. Mahesh, J. Kud, P. Dixon, Oracle at TREC8: A Lexical Approach, in: Proceedings of the Eighth Text Retrieval Conference (TREC-8), 1999.

[6] E. Voorhees, Using WordNet for Text Retrieval, in: C. Fellbaum (Ed.), WordNet: an electronic lexical database, MIT Press, 1998.

[7] J. Pound, P. Mika, H. Zaragoza, Ad-hoc object retrieval in the web of data, in: Proceedings of the 19th International Conference on World Wide Web, ACM, 2010, pp. 771–780.

[8] D. Maynard, V. Tablan, C. Ursu, H. Cunningham, Y. Wilks, Named Entity Recognition from Diverse Text Types, in: Recent Advances in Natural Language Processing 2001 Conference, Tzigov Chark, Bulgaria, 2001, pp. 257–274.

[9] L. Ratinov, D. Roth, Design challenges and misconceptions in named entity recognition, in: Proceedings of the Thirteenth Conference on Computational Natural Language Learning, Association for Computational Linguistics, 2009, pp. 147–155.

[10] D. Rao, P. McNamee, M. Dredze, Entity linking: Finding extracted entities in a knowledge base, in: Multi-source, Multilingual Inf. Extraction and Summarization, Springer, 2013.

[11] K. Bontcheva, H. Cunningham, Semantic annotations and retrieval: Manual, semiautomatic, and automatic generation, in: J. Domingue, D. Fensel, J. Hendler (Eds.), Handbook of Semantic Web Technologies, Springer, 2011, pp. 77–116.

[12] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, S. Hellmann, DBpedia – a crystallization point for the web of data, Journal of Web Semantics: Science, Services and Agents on the World Wide Web 7 (2009) 154–165.

[13] V. Tablan, I. Roberts, H. Cunningham, K. Bontcheva, GATECloud.net: a Platform for Large-Scale, Open-Source Text Processing on the Cloud, Philosophical Transactions of the Royal Society A: Mathematical, Physical & Engineering Sciences 371 (1983) (2013) 20120071.

[14] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, GATE: an Architecture for Development of Robust HLT Applications, in: Proc. of the 40th Annual Meeting on Association for Computational Linguistics, ACL'02, Philadelphia, USA, 2002, pp. 168–175.

[15] A. Kiryakov, OWLIM: balancing between scalable repository and light-weight reasoner, in: Proceedings of the 15th International World Wide Web Conference (WWW2006), 23–26 May 2010, Edinburgh, Scotland, 2006.

[16] J. Broekstra, A. Kampman, F. Van Harmelen, Sesame: A generic architecture for storing and querying RDF and RDF Schema, in: The Semantic WebISWC 2002, Springer, 2002.

[17] P. Boldi, S. Vigna, MG4J at TREC 2005, in: E. M. Voorhees, L. P. Buckland (Eds.), Proceedings of the Fourteenth Text REtrieval Conference (TREC 2005), 15–18 November 2005, Vol. 500 of Special Publications, NIST, 2005, pp. 266–271, `http://mg4j.dsi.unimi.it/`.

[18] E. Hatcher, O. Gospodnetic, Lucene in Action (In Action series), Manning Publications Co., Greenwich, CT, USA, 2004.

[19] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, M. Gatford, Okapi at trec-3, NIST SPECIAL PUBLICATION SP (1995) 109–109.

[20] K. S. Jones, A statistical interpretation of term specificity and its application in retrieval, Journal of documentation 28 (1) (1972) 11–21.

[21] H. Bast, F. Bäurle, B. Buchhold, E. Haussmann, Broccoli: Semantic full-text search at your fingertips, CoRR abs/1207.2615.

[22] B. Shneiderman, The eyes have it: a task by data type taxonomy for information visualizations, in: Proceedings of the IEEE Symposium on Visual Languages, 1996, pp. 336–343.

[23] M. Agatonovic, N. Aswani, K. Bontcheva, H. Cunningham, T. Heitz, Y. Li, I. Roberts, V. Tablan, Large-scale, parallel automatic patent annotation, in: Proceedings of the 1st ACM workshop on Patent information retrieval (PaIR '08, 30 October 2008, PaIR '08, ACM, New York, NY, USA, 2008, pp. 1–8.

[24] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, L. A. Stein, OWL web ontology language reference, W3C recommendation, W3C, http://www.w3.org/ (Feb 2004).

[25] R. G. Raskin, M. J. Pan, Knowledge representation in the semantic web for earth and environmental terminology (sweet), Computers and Geosciences 31 (9) (2005) 1119–1125.

[26] A. Spink, D. Wolfram, M. B. J. Jansen, T. Saracevic, Searching the web: The public and their queries, Journal of the American Society for Information Science and Technology 52 (3) (2001) 226–234.

[27] J. Brooke, SUS: a "Quick and Dirty" Usability Scale, in: P. Jordan, B. Thomas, B. Weerdmeester, A. McClelland (Eds.), Usability Evaluation in Industry, Taylor and Francis, 1996.

[28] S. Amer-Yahia, M. Lalmas, Xml search: languages, inex and scoring, SIGMOD Rec. 35 (4) (2006) 16–23.

[29] S. Amer-Yahia, C. Botev, J. Shanmugasundaram, Texquery: A full-text search extension to xquery, in: Proc, of the 13th Int. Conf. on World Wide Web, WWW '04, 2004, pp. 583–594.

[30] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, Xquery 1.0: An xml query language (second edition), Tech. rep. (14 December 2010).

[31] J. X. Yu, L. Qin, L. Chang, Keyword search in databases, Synthesis Lectures on Data Management 1 (1) (2009) 1–155.

[32] T. Tran, P. Mika, H. Wang, M. Grobelnik, Semsearch'11: The 4th semantic search workshop, in: Proceedings of the 20th International Conference Companion on World Wide Web, WWW '11, ACM, 2011, pp. 315–316.

[33] F. Giunchiglia, U. Kharkevich, I. Zaihrayeu, Concept search, in: The Semantic Web: Research and Applications, 6th European Semantic Web Conference, Vol. 5554 of Lecture Notes in Computer Science, Springer, 2009, pp. 429–444.

[34] M. Fernández, I. Cantador, V. López, D. Vallet, P. Castells, E. Motta, Semantically enhanced information retrieval: An ontology-based approach, Web Semantics 9 (4) (2011) 434–452.

[35] P. Chubak, D. Rafiei, Efficient indexing and querying over syntactically annotated trees, PVLDB 5 (11) (2012) 1316–1327.

[36] S. Bird, Y. Chen, S. Davidson, H. Lee, Y. Zheng, Designing and evaluating an xpath dialect for linguistic queries, in: Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on, 2006, pp. 52–52.

[37] L. Ding, T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. C. Doshi, J. Sachs, Swoogle: A Search and Metadata Engine for the Semantic Web, in: Proceedings of the Thirteenth ACM Conference on Information and Knowledge Management, 2004.

[38] Y. Lei, V. Uren, E. Motta, Semsearch: A search engine for the semantic web, in: S. Staab, V. Svátek (Eds.), Managing Knowledge in a World of Networks, Vol. 4248 of Lecture Notes in Computer Science, Springer Berlin, Heidelberg, 2006, pp. 238–245.

[39] M. Hildebrand, J. van Ossenbruggen, J. Hardman, /facet: A Browser for Heterogeneous Semantic Web Repositories, in: Proceedings of the 5th International Semantic Web Conference, 2006.

[40] G. Klyne, J. Carroll, Resource description framework (RDF): Concepts and abstract syntax, W3C recommendation, W3C, available at http://www.w3.org/TR/rdf-concepts/ (2004).

[41] K. Balog, P. Serdyukov, A. P. de Vries, Overview of the trec 2010 entity track, in: E. M. Voorhees, L. P. Buckland (Eds.), Proceedings of The Nineteenth Text REtrieval Conference, TREC 2010, Gaithersburg, Maryland, USA, November 16-19, 2010, 2010.

[42] R. Blanco, P. Mika, S. Vigna, Effective and efficient entity search in rdf data, in: Proceedings of the 10th International Conference on The Semantic Web, ISWC'11, Springer-Verlag, 2011, pp. 83–97.

[43] C. Rocha, D. Schwabe, M. P. Aragao, A hybrid approach for searching in the semantic web, in: Proceedings of the World Wide Web Conference, 2004.

[44] G. Stoilos, B. C. Grau, I. Horrocks, How incomplete is your semantic web reasoner?, in: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI), 2010.

[45] G. Zenz, X. Zhou, E. Minack, W. Siberski, W. Nejdl, From keywords to semantic queries - incremental query construction on the semantic web, Web Semantics 7 (3) (2009) 166–176.

[46] B. Popov, A. Kiryakov, A. Kirilov, D. Manov, D. Ognyanoff, M. Goranov, KIM – Semantic Annotation Platform, in: 2nd International Semantic Web Conference (ISWC2003), Springer, Berlin, 2003, pp. 484–499.

[47] N. Bikakis, G. Giannopoulos, T. Dalamagas, T. Sellis, Integrating keywords and semantics on document annotation and search, in: R. Meersman, T. Dillon, P. Herrero (Eds.), On the Move to Meaningful Internet Systems, Vol. 6427 of Lecture Notes in Computer Science, Springer, 2010, pp. 921–938.

[48] L. Zhang, Q. Liu, J. Zhang, H. Wang, Y. Pan, Y. Yu, Semplore: An ir approach to scalable hybrid query of semantic web data, in: The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, 2007, pp. 652–665.

[49] D. Damljanovic, M. Agatonovic, H. Cunningham, K. Bontcheva, Improving habitability of natural language interfaces for querying ontologies with feedback and clarifcation dialogues, Web Semantics: Science, Services and Agents on the World Wide Web 19 (0), http://bit.ly/JWSsd.