Entity-based keyword search in web documents

Enrico Sartori[1], Yannis Velegrakis[1], and Francesco Guerra[2]

[1] University of Trento
{sartori.enrico@gmail.com, velgias@unitn.eu}
[2] University of Modena e Reggio Emilia
{francesco.guerra@unimore.it}

Abstract. In document search, documents are typically seen as a flat list of keywords. To deal with the syntactic interoperability, i.e., the use of different keywords to refer to the same real world entity, entity linkage has been used to replace keywords in the text with a unique identifier of the entity to which they are referring. Yet, the flat list of entities fails to capture the actual relationships that exist among the entities, information that is significant for a more effective document search. In this work we propose to go one step further from entity linkage in text, and model the documents as a set of structures that describe relationships among the entities mentioned in the text. We show that this kind of representation is significantly improving the effectiveness of document search. We describe the details of the implementation of the above idea and we present an extensive set of experimental results that prove our point.

1 Introduction

Most search engines for documents and news on the web are powered by a keyword-based indexing system. These indexing systems model the documents as a vector, i.e., a flat list of keywords. Keyword-based search is then achieved by looking at the relative frequencies of the various keywords in the documents and comparing them with the keywords in the user query, which is also modelled as a vector. Unfortunately, this approach is prone to the ambiguities met in natural languages, for instance, the use of different terms to describe the same real world entity. This is not simply due to the use of synonym terms, but extends to even keywords that are highly different. For instance, the term “Obama” and “US president” most likely refer to the same person, despite the fact that the two terms have no semantic similarity as words. Furthermore, it is also common the case in which the same keyword is used in different situations to describe highly different things. To cope with this problem, many systems employ entity linkage, i.e., the identification of the entity to which one or more consecutive keywords are referring and the replacement of these keywords with a unique reference to the entity. After this task a document can be seen as a vector of entities or a mixture of entities and keywords.

Although the use of entities is significantly improving the accuracy of the search [20], there is still a major limitation: the loss of the relationships among the entities. Each entity in the vector representation of the document is seen independently of the others and its relationship is the same with any other entity in the vector. We advocate that by not seeing documents as a flat list of entities (and other keywords) but as a more structured whole, can lead to significantly better search results, and there is a need for a model that can represent and take this information into consideration.

In this work we provide a solution with a model that sees the document as a set of simple structures with no more than two entities each. Of course one can construct a complex structure like a graph or a tree [16] but then the complexity of the matching task is getting significantly higher. It is our belief that small structures specifying the relationships between entities found in a close proximity in the document alongside the kind of relationship that they have between them, provide the required additional information to achieve a satisfactory document search. Thus, the contributions of our work are as follows:

1. We introduce a model that instead of a vector of terms it uses a set of triples for representing documents and facilitating document search.
2. We illustrate how raw documents and user queries can be converted and modeled into that model.
3. We present an indexing mechanism that can effectively and efficiently index the structures we propose and as a consequence the documents themselves.
4. We propose a similarity metric that is used to identify the documents that are related to a given keyword query, and
5. We present a number of experimental results that demonstrate the efficiency and effectiveness of our proposed solution.

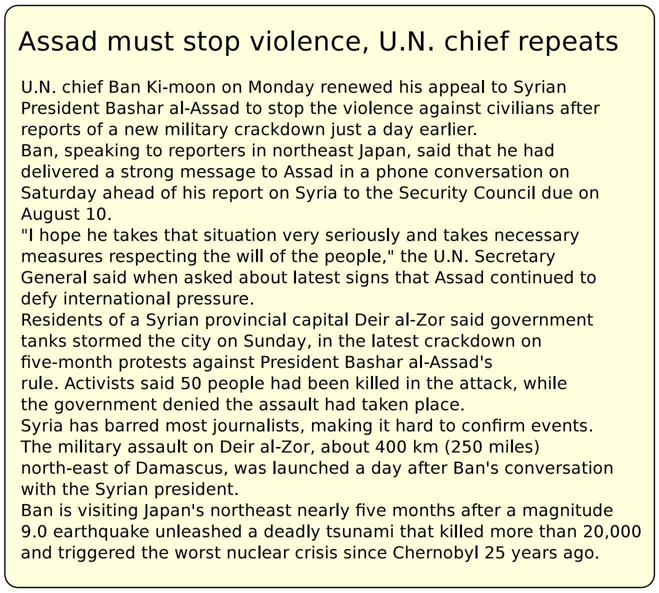
The remainder of this paper is structured as follows. Section 2 presents a motivating example that aims at helping the reader understand the importance of our approach and how it overcomes some of the limitations of the keyword search in documents. In section 3 we define formally the problem we aim to address, while in section 4 we explain in full details our solution. This includes our representation model, the indexing mechanism, the similarity formula, and the query answering algorithm. Section 5 describes the technical details of a prototype we have developed. Section 6 presents the related work and explains how we differ with what already exists. Finally, Section 7 contains the results of our experiments.

2 Motivating Example

Consider a user who is looking for a document talking about the position of the U.N. chief with respect to the situation in Syria and specifically its president Assad.

The document illustrated in Figure 1 is an article that clearly talks about the topic that the user is looking for since the two main personalities, Ban Ki Moon who is the U.N. chief and Assad, the Syrian president, appear in the document. Thus, the document would be expected to be among the results of the user query asking for the documents related to these two persons.

The challenging question to be answered, is how related the document is to the topic. Clearly the more a document mentions the topics of interest, in our case the two personalities, the more related it is. However, as it can be seen the two personalities are mentioned with different names, or with different expressions. For instance, Ban Ki


Assad must stop violence, U.N. chief repeats

U.N. chief Ban Ki-moon on Monday renewed his appeal to Syrian President Bashar al-Assad to stop the violence against civilians after reports of a new military crackdown just a day earlier.
Ban, speaking to reporters in northeast Japan, said that he had delivered a strong message to Assad in a phone conversation on Saturday ahead of his report on Syria to the Security Council due on August 10.
"I hope he takes that situation very seriously and takes necessary measures respecting the will of the people," the U.N. Secretary General said when asked about latest signs that Assad continued to defy international pressure.
Residents of a Syrian provincial capital Deir al-Zor said government tanks stormed the city on Sunday, in the latest crackdown on five-month protests against President Bashar al-Assad's rule. Activists said 50 people had been killed in the attack, while the government denied the assault had taken place.
Syria has barred most journalists, making it hard to confirm events.
The military assault on Deir al-Zor, about 400 km (250 miles) north-east of Damascus, was launched a day after Ban's conversation with the Syrian president.
Ban is visiting Japan's northeast nearly five months after a magnitude 9.0 earthquake unleashed a deadly tsunami that killed more than 20,000 and triggered the worst nuclear crisis since Chernobyl 25 years ago.

Fig. 1. An example document

Moon is mentioned some times with his name and some times as ‘U.N. Chief” or as “U.N. Secretary General”. If the search is based only on the exact keywords that the user has used in the query, the relativeness of the document will depend on the frequency in the document of the keyword that was chosen to refer to a specific entity. To avoid these there is a need to identify and treat equally all the different expressions that refer to the same entity.

Furthermore, a different factor that significantly affects the relativeness of the document to the topic for which the user is looking, is the relationship between the two items of interest. In the example document, one can see how Ban Ki Moon and Assad appear often in the same sentence, this is a clear indication that in this article these two person are in a strong relationship. The last sentence, instead, references Japan, because the U.N. chief was there at the moment of his interview. Clearly, the relationship between Assad and Ban Ki Moon is much stronger than between Assad and Japan, even if they appear in the same document. Using a vector that treats all the keywords equally, assuming the same frequency, the strength between the two personalities of interest will be considered the same as the strength between the U.N. chief and the Japan. Thus, instead of only recognising entities in the document, it is important to understand the relationships that exist between them as mentioned in the document, and realise the strength of these relationships.

3 Problem Statement

Real world documents are sequences of keywords (or words for simplicity). To model a real world document we assume axiomatically the existence of a countable infinite set of *keywords* $\mathcal{W}$. Punctuation marks are also considered keywords.

Definition 1. *A* raw document *D is a finite sequence of keywords, i.e., $D=\langle k_1, k_2, \ldots, k_n\rangle$, where $k_i \in \mathcal{W}$, for $i= 1..n$, and $n \in \mathbb{N}^*$.*

From the semantic point of view, real world documents contain statements about real world entities. Statements describe characteristic attributes that entities have, actions the entities perform on other entities, or states in which they are. The actions or the states are typically expressed by verbs. To model these verbs, we consider a countable infinite set $\mathcal{V}$ of verb identifiers.

Entities, on the other hand, are referenced by noun phrases.[3] Since different noun phrases may refer to the same real world entity, it is often preferred to use a unique identifier for referencing the entity. Note that not all the noun phrases can be replaced by an entity identifier. For instance, for an expression "a red plane", or "an article" it is clear they talk about an entity but it is not clear what is the exact entity to which they are referring. To model this, we assume the existence of two countable infinite sets of identifiers, one $\mathcal{O}$ of entities, and one $\mathcal{N}$ for noun phrases. The set $\mathcal{O}$ contains one unique identifier for every real world entity. As a set $\mathcal{N}$ we consider the set of all the possible sequences of alphabet letters, digits and symbols. In this way, there is an easy way to find the identifier of a noun phrase. It only needs to create a string from the concatenation of the words in the noun phrase and use this concatenation as the noun identifier for the noun phrase.

We also consider the existence of a special identifier "*null*" that is the only identifier that at the same time is both an entity, a verb and a noun identifier, i.e., $null \in \mathcal{O}$, $null \in \mathcal{V}$, $null \in \mathcal{N}$ and $\mathcal{O} \cap \mathcal{V} \cap \mathcal{N} = \{null\}$. The $null$ identifier functions differently than the way nulls function in databases. In particular any equality comparison of a null with another identifier or another null is always true. In other words, the null identifier functions like a wildcard.

Definition 2. *A* statement *is a triple $\langle s, p, o\rangle$, where $s, o \in \mathcal{O} \cup \mathcal{N}$ and $p \in \mathcal{V}$.*

We will use the symbol $\mathcal{S}$ to refer to the set of all possible statements.

Recent studies [8] on Wikipedia documents have indicated that considering only the entities mentioned in a document and the relationships between them preserves enough information to communicate with adequate accuracy the general semantic message of a document. Based on this, we also believe that entities (expressed through entity identifiers or noun phrase identifiers) and the relationships between them or their states (expressed through verb identifiers), provide adequate information to communicate the semantics of the content of a real world document to be successfully identified as related to the queries that ask for its information. For this reason, we define documents to be sets of statements.[4]

[3] The meaning of a "noun phrase" is the one used in linguistics

[4] Note that a "raw document" is what we defined as the document that the user provided, while a "document" is a set of statements containing identifiers and verbs

Definition 3. *A* document *is a finite set of statements, i.e., $D \subset \mathcal{S}$ and is finite.*

We will use the symbol $\mathcal{D}$ to refer to the set of all possible documents.

Similar to real world documents, a user query is a finite list of keywords, only that its length, i.e., the number of keywords of which it consists, is significantly smaller than those of the documents. Thus, we can also consider the user queries as raw documents, and define the concept of a *query* as a document.

For simplicity, in what follows, and when there is no risk of confusion, we will drop the term "identifier" when talking about a verb, an entity, or a noun, respectively.

The problem we would like to to solve is as follows. Given a collection of documents C^D and a user query q, provide an ordered list of documents in the collection, in a way that the first document is the one that is believed to be the most related to the query q, the second is the second most related, etc.

4 Solution

4.1 Processing the Documents

To convert the raw documents, i.e., the flat list of keywords of which the real world documents consist, into documents, i.e., set of statements, the very first step is to identify the keywords that refer to entities and replace them with the respective entity identifiers. This is achieved through a Named Entity Recognition process. Name Entity recognition is an extensively studied task and orthogonal to the scope of this work. Having identified the entities, the next step is to also identify the verbs. This is also orthogonal to our scope but can be achieved through syntactic and grammatic analysis, or a more complez Natural Language Processing in general, of the raw document text. After the verbs have been identified, they are replaced by the respective verb identifier for that verb. As verb identifier of a verb we consider its infinitive form. A syntactic and grammatic analysis is providing not only the verbs but also the noun phrases which are also replaced by their respective identifier. As a noun identifier for a noun phrase, it is considered the concatenation of the words that form the noun phrase, as already mentioned in the previous section. In this way the identification generation guarantees that two different noun phrases consisting of the same words but in different order, will be assigned a different noun identifiers. Any other keyword in the raw text that has not been identified as a verb, a noun or an entity, and is not a punctuation mark is eliminated.

The raw document has now been converted into a sequence of entity, verb, noun identifiers and punctuation marks. To convert the sequence into a set of statements, it is first segmented into sentences using the punctuation marks, which are then also eliminated. Every segment generated from the segmentation is a sequence of entity, verb and noun identifiers.

Since the verbs are the components that typically specify an action or a state, the verb identifiers are those that drive the statement generation process. In particular, for every verb identifier p in a segment we look at the entity or noun identifiers that appear on its left, i.e., before the verb identifier p in the sequence, and those on its right, i.e., those appearing in the sequence after the verb identifier. For each identifier s among those before, and every identifier o among those after, the statement $\langle s, v, o \rangle$ is created.

Since every segment corresponds to a sentence in the raw text, only one verb identifier will be typically present in every segment. However, since there are sentences in natural language with more than one verb, there are also segments with more than one verb identifiers. To cope with this case, when considering the noun and entity identifiers before the verb identifier p, we consider only those for which there is no other verb identifier between them and the p. Respectively, when considering the noun and entity identifiers after the verb identifier p, we consider only those for which there is no other verb identifier between them and the verb identifier p. As an example, consider the segment $\langle n_1, n_2, e_2, v_4, e_5, n_6, v_7, e_8, v_9, n_{10}, n_{11}, v_{12} \rangle$, where the v, e and n are meant to be verb, entity and noun identifiers, respectivelly. The statements that will be created containing the verb identifier v_7 with be the $\langle e_5, v_7, e_8 \rangle$ and the $\langle n_6, v_7, e_8 \rangle$, but not the $\langle e_2, v_7, e_8 \rangle$ because between the e_2 and the v_7 there is the v_6.

Of course there are special cases in which a statement as described above may not be created because the verb identifier is at the end or the beginning of the sequence or because there are two consequtive verbs. An example of such a situation is the v_{12} in the sequence above. For these cases, we use the special $null$ identifier in the place of those missing. In the particular example of v_{12}, the statements that will be created are the $\langle n_{10}, v_{12}, null \rangle$ and the $\langle n_{11}, v_{12}, null \rangle$.

At the end of this step, the raw documents have become documents, i.e., the sequences of keywords have become sets of statements. The above steps of processing a raw document are illustrated in Algorithm 1.

4.2 Processing the User Query

Similar to the raw documents, the user queries have also to be brought into the document form. Recall that the user queries are flat lists of keywords, i.e., they are like raw documents. This means that a procedure similar to the one followed for the raw documents can also be followed here.

The first step that is performed is the identification of the verbs. Keyword queries typically have no verbs, but even if they have they are in some simple form. Thus, a simple lookup to a list of verbs is enough to identify if one or more keywords correspond to verbs, and replace them by the respective verb identifier.

The next step is the identification in the query of those keywords (or consecutive keywords) that refer to some real world entity and replace them with the respective entity identifier, if possible. This can be performed through a Named Entity Recognition task.

Next, we identify the noun phrases. Natural language analysis will not perform well in general here because of the brevity of the keyword queries and the lack of a complete syntax and grammar conformity. However, the only thing needed is to identify the noun words, which can be done without the full power of natural language processing. We can, for instance, simply do a lookup in a dictionary to identify if a word is a noun or not, or run the grammatical part only of a natural language processing. Apart from the verbs and the entity identifier that we already have, every non-noun word identified is ignored. Each noun words is then considered a noun phrase, and is replaced with its respective noun identifier.

Having converted the keyword query into a sequence of identifiers (verb, entity or noun), it is now possible to create a representation of it as a set of statements. The algorithm used for this purpose is the one used in documents as presented in the previous section. However, there is a small issue to be taken care, which introduces a a small variation to the algorithim. Since it was not possible to run a full natural language analysis on the keyword query but we only characterized the keywords independently as nouns or not, there is the risk that that two or more concequtive keywords that have been characterized as nouns, should have been considered not seperatelly, but should form together a noun phrase. For instance, if the keyword query had the words "company restaurant", the partial natural language analysis that we can run would have only identify them as nouns. Then, according to the algorithm, each one should form a noun phrase, and be replaced by the respective noun identifier. However, it may be better the two nouns to be considered together as a single noun phrase. Unfortunately, we do not have enough information to decide which of the alternatives is the right one, so we consider them all during the statement generation phase. In the particular example, we would have considered three noun identifiers instead of two: the one for the "company", the one for "restaurant" and an additional one for the "company restaurant". Basically, if there are n concequtive keywords characterized as nouns, we consider all the $\frac{n*(n+1)}{2}$ ordered subsequences, and for each one we generate a noun identifier. Recall that the noun identifier for a sequence of work is the one generated by the concatenation of the words in the sequence. Each of the different noun identifiers we generate in this step is then used in the statement generation algorithm as if the respective noun phrase was present.
As an example, consider the user query:

$$q = \langle w_1,\ w_2,\ w_3,\ w_4,\ w_5,\ w_6 \rangle \tag{1}$$

which after the assignment of the identifiers becomes the sequence:

$$\langle n_1,\ n_2,\ v_3,\ n_4,\ v_5,\ n_6 \rangle \tag{2}$$

During the generation of the statements that involve the verb identifier v_3, the statements that will be generated will be the $\langle n_1, v_3, n_4 \rangle$, the $\langle n_2, v_3, n_4 \rangle$, and the $\langle n_{1-2}, v_3, v_4 \rangle$, where the noun identifier n_{1-2} is the one corresponding to the noun phrase consisting of both keywords w_1 and w_2.

A special case that is important to mention here is the case in which the user query contains no verb, i.e., the phase that turns it into a sequence of identifiers produces a sequence of only entity and noun identifiers. In such a case we generate statements that have the special verb identifier "null". Since, in the absence of the verb, we are not sure what is the relationship between the entity and noun identifiers in the sequence, we generate one statement for every possible position among the keywords in the user query that a verb could have been present. For instance, if the user query was the

$$q = \langle w_1,\ w_2,\ w_3 \rangle \tag{3}$$

that after the identification assignment had become the sequence

$$q = \langle e_1,\ n_2,\ e_3 \rangle \tag{4}$$

the statements that would have been created are: $\langle null, null, e_1 \rangle$, the $\langle null, null, n_2 \rangle$, the $\langle null, null, e_3 \rangle$, the $\langle e_1, null, n_2 \rangle$, the $\langle e_1, null, e_3 \rangle$, the $\langle n_2, null, e_3 \rangle$, the $\langle e_1, null, null \rangle$, the $\langle n_2, null, null \rangle$, and the $\langle e_3, null, null \rangle$.

The steps described above are also explained in pseudocode in Algorithm 2.

4.3 Document and Query Matching

The main step of our work is to match the query to the documents and measure how related each document is to the query. Having both the documents and the query modeled as a set of statements, any kind of similarity across sets can be used. This would work well if every statement is treated as a monolithic and undivided object. However, this is not the case, or at least not the desired behavior for our approach. Imagine a user query that asks about Obama's visit to Middle East. Clearly a raw document that talks about such a visit is highly related. Although, a raw document that talks about Obama's visit to Russia, may not be what the user is looking for, it is not completely unrelated, since it is about Obama and also about a visit of him somewhere. Similarly, a raw document talking about some action Obama took in his government, is clearly less related, yet not completely irrelevant. The above mean that in addition to the set similarity metric that we need to employ, we have to also consider the partial matching statements. For this, given a document d and a statement t of the query q, we consider three sets of statements: The first set $\mathcal{A}$ consists of all those statements in the document that fully match some statement in the query q, i.e., the exact same statement appears in the query. The set $\mathcal{B}$ consists of all the statements in the document for which there is a statement in the query that matches two of their three components, i.e., either the subject and object, of the verb and one of the subject or object. The third set $\mathcal{C}$ consists of the document statements for which there is a statement in the query with which they match only one component, the subject, the verb, or the object.

Given these three sets, we compute the following three values:

$$s_1 = \frac{|\mathcal{A}|}{|S|};\ s_2 = \frac{|\mathcal{B}|}{|S| - |\mathcal{A}|};\ s_3 = \frac{|\mathcal{C}|}{|S| - (|\mathcal{A}| + |\mathcal{B}|)} \tag{5}$$

where the set S is the set of all the statements in the document and the lines around the set name indicate the cardinality of it, i.e., the number of statements it contains. The first number indicates the percentage of the document statements that fully match. The second number indicates the percentage of the partially matching statements with two components among those that are not matching fully. The third and last number indicate the percentage of the matching document statements of the document on one component with respect to the statements that are neither fully matched, nor partially matched with two elements.

With these three numbers we can compute a score of the matching of the statement t to the document d. The idea is that the percentage of the matching statements should count the most, those that are matching with only two components, less and those matching with only one component the least. Based on that principle, and a factor s which is between 0,5 and 1, we define the score of the document d to the statement t as:

$$s(t, d) = s_1 + (1 - s_1) \times [s_2 + (1 - s_2) \times s_3] \tag{6}$$

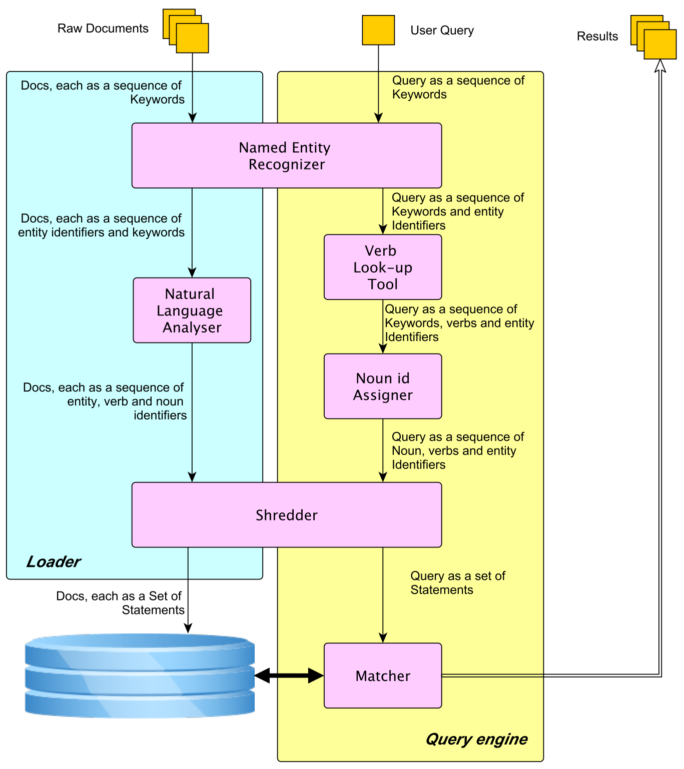

Raw Documents
User Query
Results
Docs, each as a sequence of Keywords
Query as a sequence of Keywords
Named Entity Recognizer
Docs, each as a sequence of entity identifiers and keywords
Query as a sequence of Keywords and entity Identifiers
Verb Look-up Tool
Natural Language Analyser
Query as a sequence of Keywords, verbs and entity Identifiers
Noun id Assigner
Docs, each as a sequence of entity, verb and noun identifiers
Query as a sequence of Noun, verbs and entity Identifiers
Shredder
Loader
Query as a set of Statements
Docs, each as a Set of Statements
Matcher
Query engine

Fig. 2. The Architecture of the Query Evalaluation System

Many other formulas can be used for the computation of the $s(t, d)$, however, the above is the traditional weighted sum of two factors (with weights to give a sum to 1), but extended to capture three factors accordingly.

The final relatedness score of the document d to the quert q is the average of the matching scores of the query individual statements for that document, i.e.,

$$score(d, q) = \frac{\Sigma_{t \in q} s(t, d)}{|q|} \tag{7}$$

where $|q|$ denotes the number of statements in the query q.

The query evaluation steps all algorithmically explained in pseudocode in Algorithm 3.

5 Implementation

We have materialized the previously described framework into a system, the architecture of which is illustrated in Figure 2. The figure illustrates clearly the different components and the group they belong depending on their role. In short, there is the part

Algorithm 1: Document Generation from Raw Text
Input: Raw Document D_r: $\langle k_1, k_2, .., k_n \rangle$
Output: Documemt D: Set of $\langle s, v, o \rangle$

```
DocImport($D_r$)
(1)    (Set os Statements) $D \leftarrow \emptyset$
(2)    // Recognize entities and replace the keywords with the entity identifier
(3)    (Keyword Sequence) $D_e \leftarrow NamedEntityRecognition(D_r)$
(4)    // Syntactic/Grammatical Analysis of the text and annotation of the various components.
(5)    (Keyword Sequence) $D_l \leftarrow NaturalLanguageAnalysis(D_e)$
(6)    (Set of Keyword Sequences) $F \leftarrow SegmentIntoSentences(D_l)$
(7)    foreach $D_s \in F$
(8)        for k=1 to $|D_s|$
(9)            v $\leftarrow D_s[k]$
(10)           if ($v \notin \mathcal{V}$)  continue
(11)           // Find the first verb on the left and the right of the verb v
(12)           $lb \leftarrow 0$
(13)           for i=1 to k-1
(14)               if ($v \in \mathcal{V}$)  $lb \leftarrow i$
(15)           $la \leftarrow |D_s| + 1$
(16)           for i=$|D_s|$ to k+1
(17)               if ($v \in \mathcal{V}$)  $la \leftarrow i$
(18)           // Combine every id the left with every id from the right to form a statement
(19)           for i=lb+1 to k-1
(20)               for j=k+1 to la-1
(21)                   $D \leftarrow D \cup \{ \langle D_s[i], v, D_s[j] \rangle \}$
(22)           // Special Cases
(23)           if $|D_s|$=1
(24)               $D \leftarrow D \cup \{ \langle null, v, null \rangle \}$
(25)               continue
(26)           if ($k$=1 $\wedge$ $le-k$>1)
(27)               for j=k+1 to la-1
(28)                   $D \leftarrow D \cup \{ \langle null, v, D_s[j] \rangle \}$
(29)           if ($k$=$|D_s|$ $\wedge$ $k-lb$>1)
(30)               for j=lb+1 to k-1
(31)                   $D \leftarrow D \cup \{ \langle D_s[j], v, null \rangle \}$
(32)       return ($D$)
```

called Loader that is responsible for storing the documents into the document repository after turning each one of them from a sequence of keywords that is initially, into a set of statements. The figure illustrates the various components that are involved, which also reveals the flow of the process. The Loader is used offline, when new documents are to be added into the system, and in short, implements the task described in Algorithm 1. Another part of the system is the repository, illustrated as a disk, which is designed to store the documents in the form of sets of statements. A thirt part of the system is the Query Engine, which is the component working at run-time. Upon receiving a user query, it converts it into a set of statements by actually implementing Algorithm 2. Once the query has been converted to a set of statements, the Query Engine invokes the Matcher subcomponent that will compare this set to the set os statements of the doc-

Algorithm 2: User Query to Set of Statements
Input: User Query Q_u: $\langle k_1, k_2, .., k_n \rangle$
Output: Query Q: Set of $\langle s, v, o \rangle$

```
QUERYPREP($Q_u$)
(1)   (Query) $Q \leftarrow \emptyset$
(2)   (Sequence of keywords and identifiers) $D_e \leftarrow NamedEntityRecognition(Q_u)$
(3)   for i=1 to $|D_e|$
(4)       if ($D_e[i] \in \mathcal{O}$)   continue
(5)       if ($lookupIfVerb(D_e[i]) \neq \emptyset$)
(6)           $D_e[i] \leftarrow$ generateVerbIdentifier($D_e[i]$)
(7)           continue;
(8)       if ($lookupIfNoun(D_e[i]) \neq \emptyset$)
(9)           $D_e[i] \leftarrow$ generateNounIdentifier($D_e[i]$)
(10)  // Replace every sequence of consequtive noun identifiers, with its powerset sequence
(11)  (Sequence of Identifiers) $T \leftarrow \emptyset$
(12)  (Sequence of Identifiers) $I \leftarrow \emptyset$
(13)  for i=1 to $|D_e|$
(14)      if ($D_e[i] \in \mathcal{O} \cup \mathcal{V}$)
(15)          if ($T \neq \emptyset$)
(16)              (Set of Idnetifier Sequences) $P \leftarrow$ Powerset($T$)
(17)              $T \leftarrow \emptyset$
(18)              $I \leftarrow I + P$
(19)          $I \leftarrow I + D_e[i]$
(20)      else
(21)          $T \leftarrow T + D_e[i]$
(22)  for k=1 to $|I|$
(23)      v $\leftarrow I[k]$
(24)      if ($v \notin \mathcal{V}$)   continue
(25)      $lb \leftarrow 0$
(26)      for i=1 to k-1
(27)          if ($v \in \mathcal{V}$)   $lb \leftarrow i$
(28)      $la \leftarrow |I| + 1$
(29)      for i=$|I|$ to k+1
(30)          if ($v \in$ V)   $la \leftarrow i$
(31)      for i=lb+1 to k-1
(32)          for j=k+1 to la-1
(33)              $Q \leftarrow Q \cup \{ \langle D_s[i], D_s[v], D_s[j] \rangle \}$
(34)      if $|I|$=1
(35)          for j=lb+1 to k-1
(36)              $Q \leftarrow Q \cup \{ \langle null, v, null \rangle \}$
(37)              continue
(38)      if ($k$=1 $\wedge$ $la-k$>1)
(39)          for j=k+1 to la-1
(40)              $Q \leftarrow Q \cup \{ \langle null, v, D_s[j] \rangle \}$
(41)      if $k$=$|I|$ $\wedge$ $k-lb$>1
(42)          for j=lb+1 to k-1
(43)              $Q \leftarrow Q \cup \{ \langle D_s[j], v, null \rangle \}$
(44)  return ($Q$)
```

Algorithm 3: Query Answering

Input: User Query Q_u: $\langle k_1, k_2, .., k_n \rangle$, Document Collection C^D
Output: Ordered List of Documents Ans

```
EVALUATE($Q_u$)
(1)    (Sequence of $\langle Document, score \rangle$) $Ans \leftarrow \emptyset$
(2)    (Set of Documents) $L \leftarrow \emptyset$
(3)    (Query) $Q$ = QueryPrep($Q_u$)
(4)    foreach $D \in C^D$
(5)        int score = $ComputeScore(D, Q)$
(6)        $Ans$ = $Ans$ + $\langle D, score \rangle$
(7)    $Ans \leftarrow OrderByScore(Ans)$
(8)    return ($Ans$)
```

uments in the repository and compute the matching scores for the stored documents, rank them, and return the ranked list of the matched documents to the user as an answer. These are the steps that Algorithm 3 describes. As with the case of the Loader, the flow illustrated in the architecture diagram for the Execution Engine reveals also the process flow.

In what follows we will provide a description of how each of these compoinents of the system has been implemented and we will explain our design choices.

5.1 Named Entity Recognition

For the named entity recognition task we have opted for an external service instead of building a native solution. In particular, we are using the OpenCalais [24] which is freely available on the Internet. It is a mature, reliable and successful solution that can support our needs, without requiring an advanced pre-training that most of the other existing solutions require. Furthermore, the system is continuously updated and is guaranteed to perform in always new environments and communities.

A disadvantage of this solution is the latencies introduced by querying an external service over the network. To limit this disadvantage we adopted a "hybrid" implementation, i.e. to store in our database the phrases used to describe the entities encountered in the dataset. Intuitively, we have made our system to work as a cache of the expressions that have been so far met in the documents and mapped to entities by the OpenCalais. This has significantly reduced the latencies of this component. Furthermore this caching has a significant by-product advantage. Once it runs for the document collection, all the entities appearing in the document will be stored in the cache. When a new query arrives and the named entity identification runs on it, if for a sequence of keywords it is found that it is not present in the cache, it means that there is no document in the collection mentioning that entity, thus the query can be answered without any further delay.

OpenCalais is accessed through a REST interface using the method:

$$Enlighten(key, content, configuration)$$

which, given the HTML representation of a document, it cleans the text from the tags, and performs the Named Entity Recognition task, returning to the caller an XML representation of the result. The response from the service contains, for each entity, a *unique*

Id, which is used by our system as well for the identification of the entity. Moreover, OpenCalais gives information about the position of the entity inside the analyzed text, and the sequence of keywords that describe it. This sequence of keywords is used in the cache entry that was previously mentioned.

Once the list of entities has been collected, the system marks the text of the document with the ids of the objects appearing into it. In practice it replaces the phrase describing the entity with the id, storing the information about the entity inside the database, to build the previously mentioned cache.

5.2 Natural Language Analysis

To identify the verbs and the nouns in the document sentcences we need to perform some natural language text analysis. For this, we employed the NLTK [22], a well known and extensively used toolkit providing support for many common task in natural language processing.

As a first step, the document is tokenized by being divided into sentences and then into words. To perform the sentences isolation we relied on the algorithms provided by the NLTK toolkit, which allowed us to divide correctly any complex text. The tokenization algorithm takes a sentence and isolates the words appearing into it, preserving the position of the tokens inside the original sentence.

After this preprocessing step every sentence in the document is represented like a flat list of tokens, sorted according to their appearance in the sentence. Over this list is then applied a Part Of Speech tagging algorithm, which can recognize and annotate the different grammatical meaning of every token in the sentence. In our test implementation we relied again on the POS tagging algorithm provided by the NLTK. The main focus in this phase of the document processing is to tell apart nouns and verbs, in fact we need to recognize verbs, and link them with the keywords or the entities appearing in the noun part of the phrase. The output of this algorithm is a list of pairs $(token, POS\ tag)$, once again maintaining the original order of the tokens in the analyzed sentence.

The POS tagging result is needed to construct a tree representing the structure of a sentence. In fact the list of POS tags is parsed and the system builds a tree of every sentence, matching the grammatical structure of the sentence. This tree is needed to resolve the link between verbs and nouns, trying to interpret the structure of the sentence, and connect the correct section of the phrase. For our purposes the aim is to obtain a tree which can be used to separate section of the phrase composed by nouns from the verbs and to follow the links between verbs and the nouns referring to it. To achieve this NLTK offers a parser, which by using as input a POS tags list can construct a tree of the sentence, querying a grammar given by the user. So we developed a grammar reflecting the structure of the information we need to extract from a natural language sentence. After this step, the sentence is in the form of a tree, representing the grammatical role of each token (noun, verb) and the relationship between them.

After the nouns and the verbs have been characterized in the text, the algorithms presented in the previous section for statement generation can be executed.

5.3 Repository Document Indexing

Since documents are sets of statements and a statement is a triple, we need a way to efficiently access the triples related to certain queries. To do so there is a need for an effective index structure. The index should help in the computation of the cardinality of the sets $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ defined before by returning the number of triples that satisfy some search parameter. In particular, in order to compute the cardinality of set $\mathcal{A}$, the index should be able to compute the following function

$$i_{\mathcal{A}}(s, v, o, d) = cnt_{s,v,d,o} \tag{8}$$

where $cnt_{s,v,o,d}$ is the number of triples belonging to document d, that have the value s as *subject*, o as *object* and v as *verb*. For the cardinality of the set $\mathcal{B}$ it needs to be able to compute the function

$$i_{\mathcal{B}}(s, o, d) = cnt_{s,o,d} \tag{9}$$

where $cnt_{s,o,d}$ is the same as $cnt_{s,v,o,d}$, but with the v being of any value. Finally, the number of triples belonging to set $\mathcal{C}$, containing all the triples of the document that have a particular *subject* or *object*, requires the computation of the function:

$$i_{\mathcal{C}}(s, o, d) = cnt_{s,d} + cnt_{o,d} \tag{10}$$

Where $cnt_{s,d}$ and $cnt_{o,d}$ count respectively the triples sharing only the *subject* with the query and only the *object*.

The index should be effective in terms of space in order to reside in main memory and achieve efficient look-ups. It should also support incremental updates when new documents are encountered by the system, without the need for a complete refresh of the whole structure through a from-scratch re-computation.

Basically our approach is derived from the Hexastore [25], we extended its algorithm and adapted it to fit our data and usage context. Hexastore is a system for the storage and querying of RDF triples. The idea behind it is to provide an indexing structure for every possible order of the three terms in a triple. It permits a fast filtering of the triples in a database, due to the ability to group the data with respect to any field in the triple.

The data structure we defined in order to implement a similar index is basically composed of a set of nested associative arrays, which can be navigated recursively using the fields of the triples as keys of the arrays. Figure 3 depicts how these nested arrays work. In this case, the structure permits to implement the function in formula 8. In fact, it is possible to follow the links from an array to the other and reach the counter associated with the last array. This field counts the number of triples in the document selected which have the *subject*, *object* and *verb* equal to the one searched. More specifically the first vector is an associative array whose keys are the set of different *subjects* encountered among all the triples. Each entry in this array is connected to a pair which is composed by a counter and another associative array. This structure is repeated recursively for the *object* field of the triple, and then for the verb. The cells of the last associative array are linked only to the respective counter.

Since our structure is fundamentally based on the Hexastore idea, the complexity of the retrieval of the respective tuples is the same to the one of Hexastore. A detailed study of its performance can be found elsewhere [25].

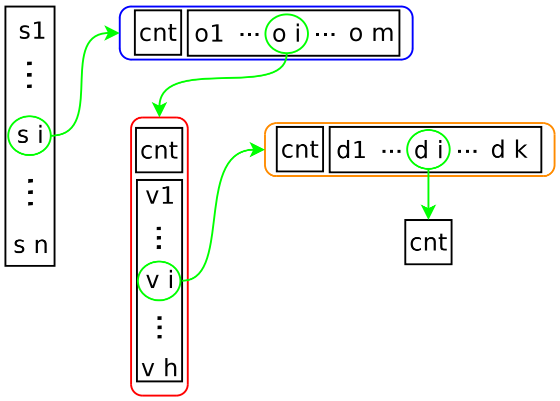


Fig. 3. Schematic representation of the index structure

To find the number of triples that match a triple over all their three fields in a document, we maintain and query a structure of the form:

$$\begin{aligned}[s_1 \dots s_n] &\rightarrow (c_s, [o_1 \dots o_m]) \rightarrow \\ &\rightarrow (c_{s,o}, [v_1 \dots v_l]) \rightarrow \\ &\rightarrow (c_{s,o,v}, [d_1 \dots d_h]) \rightarrow \\ &\rightarrow c_{s,o,v,d} \end{aligned} \tag{11}$$

The notation above means that we have an array of the different subject values ($[s_1 \dots s_n]$). For each entry in that array, i.e., for every subject s, we have a pointer to an array $[o_1 \dots o_m]$, with each element of which corresponding to an object value from all those object values met in the triples that have s as a subject (c_s). An element of that array corresponding to an object value o is a pointer to an array $[v_1 \dots v_l]$. Each element of that array corresponds to a verb v among all those verb values met in the triples that have subject s and object o. An element of the array $[v_1 \dots v_l]$ corresponding to the verb v is a pointer to an array $[d_1 \dots d_h]$, each element of which corresponds to a document in the collection, and indicates the number of triples of the form $\langle s, v, o \rangle$ that the respective document contains, i.e., the value of the function $c_{s,o,v,d}$. The latter is needed as described in formula (8) which in term is needed for the computation of the quantity $\mathcal{A}$ used in the formula (5).

In a similar fashion, to be able to quickly identify the number of triples in a document that match a specific subject and object, we maintain and use a structure of the form:

$$\begin{aligned}[s_1 \dots s_n] &\rightarrow (c_s, [o_1 \dots o_m]) \rightarrow \\ &\rightarrow (c_{s,o}, [d_1 \dots d_h]) \rightarrow \\ &\rightarrow c_{s,o,d} \end{aligned} \tag{12}$$

which materializes the function $c_{s,o,d}$, used to compute the function mentioned in formula (9) which is required for the computation of the quantity $\mathcal{B}$ used in the formula (5).

Finally, for computing the function of the formula (10), we build and implement two structures, aimed at providing the numbers of triples in each document that match either the subject only or the object only. These structures are described in the formulas (13) and (14). They are used to retrieve the counts of triples matching at least one between *subject* and *object* fields of the given triple.

$$\begin{aligned}[s_1 \dots s_n] \rightarrow (c_s, [d_1 \dots d_m]) \rightarrow \\ \rightarrow (c_{s,d})\end{aligned} \tag{13}$$

$$\begin{aligned}[o_1 \dots o_n] \rightarrow (c_o, [d_1 \dots d_m]) \rightarrow \\ \rightarrow (c_{o,d})\end{aligned} \tag{14}$$

Note that the functions $c_{s,d}$ and $c_{o,d}$, include naturally the results of $c_{s,o,d}$ which in turn includes the results of $c_{s,o,v,d}$. The query answering algorithm, which we will present in the next subsection, will have to encounter for that fact.

The four structures that were just presented form our index structure. The index is built when the system starts, and is kept in memory throughout the operation of the system. Incremental updates are easily implemented due to the nesting of the structures and the use of associative arrays for the materialization of the respective array structures.

With the use of NLTK, the different keywords in the user query are tagged with the respective grammatical role, and the information about the position inside the raw document is maintained. The order of the words inside a query is important because we assume that, generally, when a user types a query, he or she tends to put verbs near to the nouns to which they refer.

5.4 Matcher

The matcher is invoked to identify the matching documents. In order to measure the relatedness of a document with respect to a specific statement in the user query, we need to identify the statement sets $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ introduced in Section 4.3. The set $\mathcal{A}$ is built by a lookup in the structure represented in formula 11. Accessing the different associative arrays with the values of the triples fields, is possible to retrieve the list of documents containing a triple exactly like the one given. The second set is instead populated by the result of a visit in the structure in formula 12. In fact this structure returns the documents which have *subject* and *object* equal to the given triple. The counter associated with the document retrieved at this step will consider even the statements belonging to the set $\mathcal{A}$, therefore we need to subtract the previously computed result to obtain the right cardinality of the set $\mathcal{B}$. We are interested in knowing the cardinality of these sets, not in retrieving the specific statements, so we do not need to effectively isolate the statements of the set $\mathcal{B}$, but just keep track of the number of statements encountered. The computation of the cardinality of the set $\mathcal{C}$ is more complex and need two lookup in different structures. In fact here we need to count the statements that match only the *subject* or the *object* of the given statement. For the first subset the answer is provided by the structure in equation 13, while for the latter is used the formula in 14. The combination of these two subsets produce the set of statement that match the statement with *at least* one field among *subject* and *object*. In order to restrict the result only to the statements which match the statement with *only* one field it is necessary to subtract the cardinality of the previously computed sets. Now that the cardinality of each different class of similarity has been computed, it is possible to measure of relatedness of a document with respect to the statement given, using the equation in formula 6. The formula is computed for all the statements in the query, and then the results are averaged in order to compute the final measure of similarity between the query and the document. Once the results are computed, the list of documents is ranked with respect to the similarity score and returned to the user.

As previously mentioned, in our implementation, the index structures used to compute the cardinality of the three classes of similarity, return the values for all the documents containing the statement. In this way there is no need to compute the similarity for each document one by one, but for each statement is possible to retrieve the list of the documents related to that information. This approach has two important advantages: the first is that in this way we consider, and compute metrics, only for the documents related to the query, rather than analyzing one by one all the documents, considering even the ones totally unrelated to the query. The second is that by analyzing every single document independently, many statements will be compared multiple times, because they are shared among different documents.

6 Related Work

The work we present in this paper belongs to the information retrieval field, since its main objective is to provide an approach to document retrieval and ranking approaches. The first approaches appearing in this field aimed to predict the probability distribution of terms inside documents, and hence determine the probability of relevance of a document with respect to a query. A very successful IR modeling approach is to convert the document into a vector space based representation, and therefore apply common techniques from linear algebra for computing similarities and operations on the documents. Generally this vector space is built using each word appearing across the dataset as a dimension of the space. A document in this space is represented as a vector, having as value for each dimension the frequency with which the corresponding word appears in the document. Once the documents are represented as vectors is possible to compute a measure of similarity between two of them using the *cosine similarity*. A broadly adopted document representation model, relying on the vector space, is the TF-IDF approach, which weights each word appearing across the dataset in a inversely proportional way with respect to the frequency of appearance of the word. This gives more importance to rare words appearing in a document, which will bring more relevance to the document if they match the user query. Roelleke and Wang [21] give a review of many aspects related to this approach to term relevance computation. An interesting addition to this approach is LSI (Latent semantic indexing), which permits to select the most informative words inside a document, making computation easier and removing noise. Moreover, this approach can isolate words that characterize a set of documents, enabling a semantic analysis of the content of these documents. An analysis of the performances of LSI, and some improvements at the existing approach are presented by Ando and Lee [3]. Since our concept is to identify the important words (entities, verbs and nouns) and not take into consideration other parts of the document apart from those three, in some way we are similar to LSI, that prioritise the words. Recently the ways of communication on the web has seen a broad evolution, with the average user becoming an active producer of information. To adapt to new situations and exploit additional data made available by websites, novel approaches to information retrieval have been presented. The model of publishing content used for blog, in which a single page can contain several documents, opens new possibility for finding different ways to deal with the problem to find the most suitable model to represent the documents. Moreover this kind of situation can shift the problem of document searching to blog searching based on the content of its posts. Another environment which offers different possibilities to model the information collected from webpages is Wikipedia, that enriches its content with metadata and links to other resources, this topics are investigated in [4].

Another important topic interested by this work regard the techniques used for named entity and entity management. One of the problems related to information retrieval, is that often the same entity has a number of different ways in which it can be referenced, this motivates the need of techniques that can identify uniquely objects on the web. An important contribution to this topic is the OKKAM project, which aims to integrate existing information about entities on the web and provide unique identifiers for them. Bouquet et al [7] propose the use of an Entity Name System (ENS) with the aim to enable information collections to use unique identifiers for objects on the web. The use of entities can bring advantages in addressing different problems, as in data integration, where linking entities coming from different datasources is a major challenge. Ioannou et al [15] present a query answering algorithm that permit to resolve entities and link together the information referring to them coming from different datasources.

As underlined in the rest of the paper we adapted a preexisting indexing algorithm to suit our needs for a fast way to access the data. The original work by Weiss et al. [25] addresses in particular the problem of indexing RDF data. In fact they consider this format as the main approach to represent semantic data retrieved from the Web. Their work aims to overcome the difficulties in terms of efficiency and scalability of managing large RDF databases, through the implementation of an indexing structure which allows fast query processing. The index is a set of nested associative arrays, which permit to group together, and therefore search, RDF triples with respect to their values. More conventional approaches to triple storage [11] and querying attempts to store the data in relational-like structures, but complex queries involve gathering information from multiple tables, worsening the performances of the system.

Sentence summarization techniques, used in this work to identify relationships between entities inside natural language, can be performed following different approaches. The goal of this task is to provide a context graph of the sentence, a formalism used to identify the semantic role played by the words into a natural language sentence. In this field, the development of ontologies of words available over the Internet enabled the application of approaches which relies on these platforms to perform their task.

Hensman [13] present an algorithm that uses the WordNet database to construct context graphs from sentences. The approach followed is to retrieve from the ontology a set of possible phrase structure that the verb encountered can support and matching them with the sentence analyzed. Lei Zhang and Yong Yu [26], proposed a machine learning approach to address this problem, their work is focused on the context graph construction for sentence coming from a specific domain, in order to overcome the difficulties of dealing with general natural language sentences.

Understanding the semantics of the keyword query is a field that has received considerable attention [2][18][23][1]. Many works have studied the way to map the query keywords based on the database structures [14], or following some semantic approach without access to the database instance [5][6]. Many of these techniques can be used to enhance our approach, however, it is important to keep in mind that they do not actually employ any named entity identification tech niques on the user query but they try to express the semantics of the keywords in terms of database structures. In that sense, they cannot be directly applied without some significant adaptation.

On the other hand, entity recognition in query has been used [12] before in a successful way and this task has been the driving force for the semantic search engines [10]. Named entities have not been used only for document retrieval but also other related data management tasks like indexing [19] and clustering [9]. For this reason, we believe that our querying technique is a significant complement to these works.

Finally, our idea of seeing the documents in a more structured form than a simple list of keywords has been studied in the past [16], however, the specific approach uses graphs as the document representation. We believe that graphs, despite very expressive, bring a unnecessary complexity than the sets of triples that we employ.

7 Experiments

This section will describe the experimental methodologies and results obtained by testing our approach in terms of execution times, memory consumption and result quality. In section 7.1 are presented the results obtained in terms of time performance in analyzing the data, and storing the information in the database. The next experiments, illustrated in section 7.2, target the analysis of the time and memory requirements for the index structure, with respect to the dataset dimensions. Section 7.3, presents the performance of the query answering algorithm in terms of execution time. This experiment is divided into two partd, the first measures the response time with respect to the dimension of the query provided, while the second analyzes the answering time over the dimension of the dataset. Finally in subsection 7.4 we present the outcome of a comparison between a Lucene [17] search engine implementation (keyword-based) and our solution.

The dataset used to test the system is composed by 4000 documents, retrieved from newspaper websites. The documents have been collected regularly on a time window of some months, using the RSS feed of the newspaper to track the new documents published. The total size of the dataset is 321 Mb, which means that the average size of a document is 82.17 Kb, while, once the document is sent to OpenCalais, analyzed and stored into the database, its size shrinks to 23.56 Kbk and the whole dataset becomes something around 126MB. We developed a web crawler able to read the RSS feeds and retrieve the documents published, keeping memory of the documents already retrieved (through a signature) in order to avoid duplicate data in the dataset. The documents were not selected from a particular domain an in general the OpenCalais was recognizing around 2-3 entities every 4-5 sentences. The nouns of the sentences that were not recognized as entities, as explained in the previous sections remained as is. The sentences had an average of 18 words in length. Notice that there is no correct translation of the documents (or the queries) into triples. As long as the matching task is sucessfully identifying the related documents.

The system tested in this section is an implementation of the approach written in Python 2. The advantages of this choice for the language are represented by the cross-platform feature of the language and the fast prototype developing time permitted by the richness of built-in structures featured by the language. The disadvantages are mainly the high resources consumption in terms of time and space that a Python implementation brings. The system was run on a computer running Ubuntu 14.04.2, with 8GB of RAM and a 64bit, 2.40GHz 1-core CPU.

7.1 Document analysis

In this experiment is evaluated the time needed to analyze all the document in the dataset, and store the resulting data in the database. This corresponds to the first phase of the system lifecycle, and takes into consideration the time spent querying the OpenCalais service, and the analysis algorithm execution.

The plot in Figure 4 shows the time taken by the system to perform the fetching and analysis of all the documents in the dataset over the dimension of the dataset itself. The higher of the two lines measure the time needed to fetch and analyze the documents, while the second is referred only to the analysis time, which depends entirely on our algorithm. The time needed to receive a response from OpenCalais, instead, depends on a variety of factors out of our control, mainly network latencies, document length and load condition on the remote service.

The execution time of our algorithm is itself divided in some subprocesses, which concur to this phase of the main process. In fact the system needs first to read the data from the machine filesystem, than it performs a preprocessing step. During this preprocessing step the text of the document is altered in order to mark the *entities* occurrences appearing in the article, and the relative information is prepared to be stored in the database. After this the effective grammatical analysis of the text is performed, which is the most time consuming operation in this phase. Once the algorithm terminates, the results are stored in the database.

This plot shows how the execution time of the process is linearly dependent from the number of documents in the dataset, and in fact, from the experimental results evaluation, has been noticed that the time needed to analyze a single document is constant independently from the number of documents. The details of the time used to analyze a document are presented in table in Figure 5, which shows how apart from the time spent waiting for the OpenCalais response, most of the time is spent in the analysis of the document's sentences.

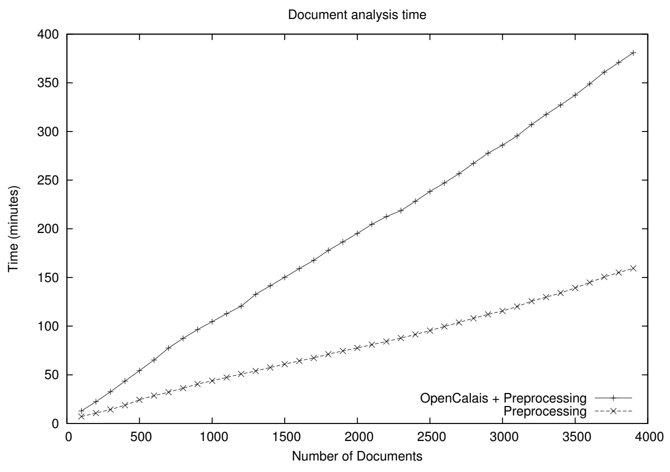


Fig. 4. Analysis time over dataset dimension

TASK	TIME
Query OpenCalais:	**3.375 s**
Read file:	0.001 s
Query service:	3.359 s
Storage time:	0.014 s
Analysis time:	**1.917 s**
Read file:	0.015 s
Entities marking:	0.004 s
Sentences analysis:	1.797 s
DB storage:	0.775 s

Fig. 5. Average Time of the individual tasks of processing a Document.

7.2 Index building

This experiment aims to show how the index, described in section 5.3, performs in terms of time needed to fully populate the structures, and the memory consumption of the complete index. These features are evaluated with respect to the dimension of the dataset used, in order to underline the behavior of the system with growing amount of data. The plot in Figure 6 shows how much time the system takes do build the index in relation with the dimension of the dataset.

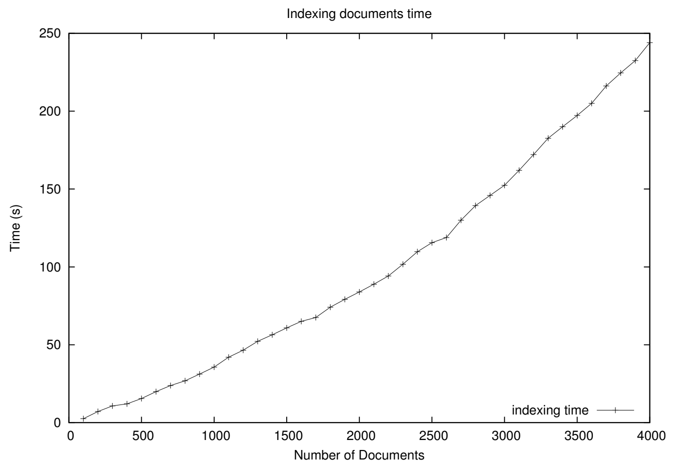


Fig. 6. Indexing time over dataset dimension

The correlation between these two features is proportional to the number of documents multiplied by a small factor, because the time taken by the system to index a single document increments as the number of documents grows. This increment is given probably by the implementation of the Python dictionaries (associative arrays), used to implement the structure, which takes more time to complete the insertion operation as the dimension of the structure grows. The time needed to index a single document is reported in Figure 7 in relation with the dataset dimension.

The other parameter which deserves to be considered in the evaluation of the index performances, is the footprint in main memory which maintaining this structure brings. As underlined in section 5.3 the structure will contain all the data retrieved from the triples appearing across all the document in the dataset. For this reason the memory needed to store a similar structure is expected to grow linearly with the number of documents analyzed. In fact the plot in Figure 8 shows this linear dependency between number of documents and memory used.

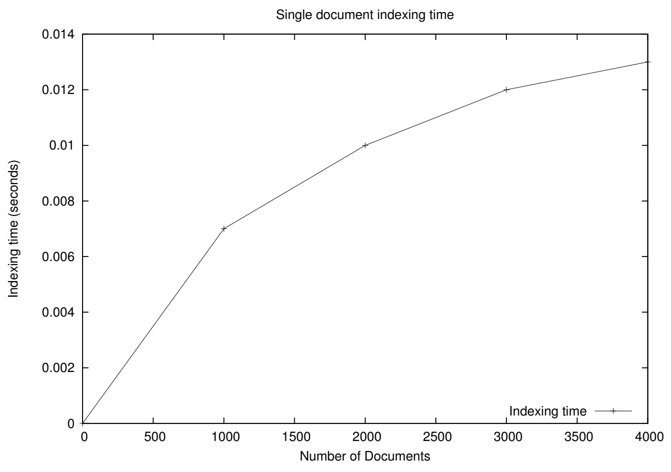


Fig. 7. Time taken to index a single document over dimension of the dataset

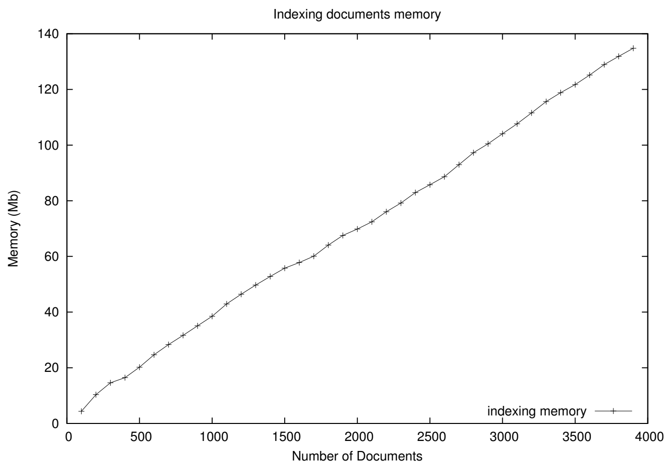


Fig. 8. Memory consumption of the index over number of documents

7.3 Query answering

The aim of this experiment is to evaluate the performance of the query answering algorithm. In order to populate a set of queries which have at least a corresponding document in the dataset, we built a system which constructs automatically queries using information contained in the dataset. In particular we tested our algorithm over three different kinds of queries, that we built following the approach described here:

- **Group 1:** The system selects a random document and retrieves the list of entities belonging to it. Form this list it extracts randomly a number (specified as a parameter) of entities. The query is than composed by concatenating a randomly chosen expression describing each entity. For our experiments we built queries with a number of entities ranging from 1 to 5.
- **Group 2:** To construct this set of queries a document is taken randomly, then a configurable number of triples is selected among the ones appearing in the document (for the experiment we used parameters ranging from 1 to 3). The content of the triples fields are than concatenated to form the query, and if one of the fields is an entity, it is replaced with an expression that corresponded to that entity (based on the information we have in the cache information from the OpenCalais).
- **Group 3:** It follows the same procedure as the previous group, with the difference that the entities are replaced not with any random expression that corresponds to them but to the one that actually appears in the selected document. For the last experiment as well we used a parameter for triples ranging from 1 to 3.

For each group of queries we run a series of experiments measuring the time needed to build the query and the total response time used by the system to provide the results. Moreover we kept track of how many triples are produced from a keyword query. These results are presented by two graphs in figures 9 and 10, one showing the mean response and query building time, and the other showing the distribution of the queries in terms of number of triples generated. The time graph shows the mean response time over the number of triple, grouped in buckets of width 50, while in the other the width of the buckets is 25.

From the graph in Figure 9 is easy to evince that the response time grows proportionally with the number of triples produced by the query, as it is expected to behave. Moreover, the system can answer queries with up to 900 triples in less than 0.5 s. The plot in 10, instead, shows how queries of different dimension (in terms of triples produced) are distributed. Is important to notice that most of the queries produce a number of triples less than 600 and after this value we encounter few queries. This helps to explain the behavior of graph 9. In fact, as the number of triples grows, the queries are fewer, and this brings to less predictable results in terms of mean response time.

7.4 Query answering quality

This section presents the outcome of the experiments run in order to measure the quality of the query answering algorithm. We implemented a test keyword-based search engine, and measured its performance against our approach, running the same queries on the two systems. The queries used are presented in Figure 11(b), and have been selected

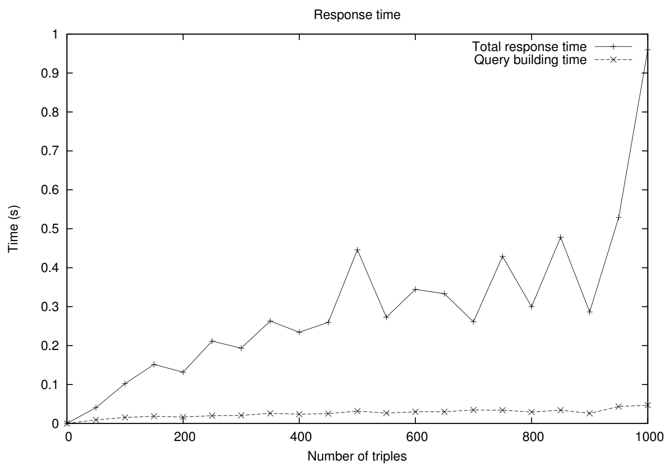


Fig. 9. Mean response time over number of triples in the query

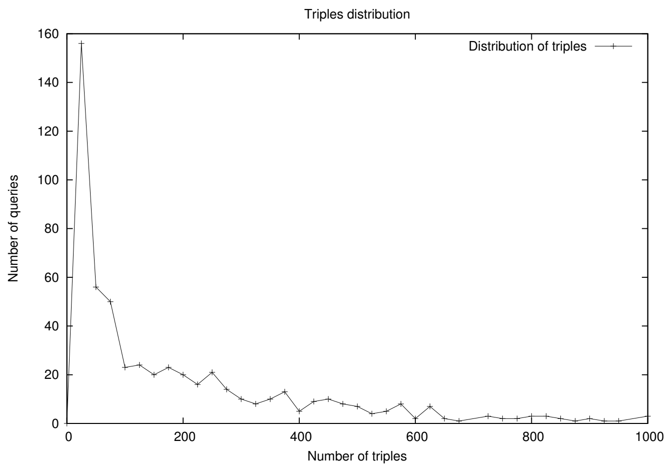


Fig. 10. Number of queries over number of triples in the query

LUCENE	BOTH	ENTITY-BASED	QUERY ID
5 (0.6 %)	675 (92 %)	47 (6 %)	1
8 (0.8 %)	926 (96 %)	30 (3 %)	2
9 (2 %)	312 (90 %)	29 (8 %)	3
4 (0.9 %)	396 (94 %)	20 (4 %)	4
34 (10 %)	267 (79 %)	38 (11 %)	5

(a)

QUERY ID	KEYWORD QUERY
1	Saudi pressure Yemen
2	Sarkozy Ghaddafi Libya
3	English Premier League
4	Fukushima reactor meltdown
5	Bhopal gas leak

(b)

Fig. 11. Query Answering Quality Evaluation Results

by hand in an effort to simulate a human user behavior. The Figure 11(a) presents the results of the quality comparison, showing in the first column the number of documents retrieved only by the keyword-based (Lucene) implementation, in the second column there is the number of documents retrieved by both systems and the third column gives the number of documents found only by our approach.

From the queries results is possible to see that often an entity-based approach can retrieve more documents than a keyword-based one. The main reason for this is that once a group of keyword is resolved into an entity, this permits to collect the document that refers to the same entity using different keywords. An example of this situation can be the keyword "*Sarkozy*", in the query no. 2, once resolved into an entity it can collect document which refers to "*president of France*". These documents, instead are not retrieved by the keyword-based approach, because they do not contain the exact word "*sarkozy*".

A different situation happens in the fifth query, which aims to retrieve documents talking about the Bhopal disaster. The word "gas", appearing in the query is very common and permits to the keyword-based approach to find many other documents, even if they are less correlated to the query. The entity based one, instead, tries to resolve the set of keywords into the entity corresponding to the event, missing some documents which appear less correlated to the query.

The results show that few documents are retrieved only by the keyword-based engine even when our approach performs better. This behavior is explained by the different way in which our approach treats verbs and entities. While in the traditional approach verbs are just keywords, and the documents containing them are retrieved, in our approach verbs are considered only in relation to the entities they refer. Documents are retrieved if they contain the entities recognized in the query, while the presence of the exact verbs is used to enhance the result of a document. The only appearance of the verb in the document, without reference to the entities is not sufficient to consider the document relevant.

8 Conclusion

In this work we presented a different approach to document representation and query answering. The main features of the techniques we developed are the representation of a document in term of relationships individuated inside the text, and the resolution of groups of keywords in uniquely identified entity. This permits on one side to deduplicate the data, through the entity resolution, and the relationships based representation can increase the expressive power of the document, maintaining information about the actions performed by the entities appearing in the article. The experiments conducted on our testing implementation of this approach show how it scales in terms of time and memory consumption, and they prove the possibility to adopt a similar system to manage large datasets of documents. One of the possible environments of application of this work, as pointed out across the rest of the paper, is in fact the possible use inside a search engine for web documents, where the requirement of keeping a low response time with large datasets has a relevant importance.

References

[1] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, Parag, and S. Sudarshan. Banks: Browsing and keyword searching in relational databases. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 1083–1086, 2002.

[2] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 5–16, 2002.

[3] R. K. Ando and L. Lee. Iterative residual rescaling: An analysis and generalization of lsi. *Proceedings of the 24st annual international ACM SIGIR conference on Research and development in information retrieval*, 2001.

[4] J. Arguello, J. L. Elsas, J. Callan, and J. G. Carbonell. Document representation and query expansion models for blog recommendation. *Association for the Advancement of Artificial Intelligence Conference*, 2008.

[5] S. Bergamaschi, E. Domnori, F. Guerra, R. Trillo-Lado, and Y. Velegrakis. Keyword search over relational databases: a metadata approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 565–576, 2011.

[6] S. Bergamaschi, F. Guerra, M. Interlandi, R. Trillo-Lado, and Y. Velegrakis. Combining user and database perspective for solving keyword queries over relational databases. *Information Systems*, 2016(55):1–19, 2016.

[7] P. Bouquet, H. Stoermer, C. Niederee, and A. Mana. Entity name system: The backbone of an open and scalable web of data. *Proceedings of the IEEE International Conference on Semantic Computing*, pages 554–561, 2008.

[8] S. Bykau, F. Korn, D. Srivastava, and Y. Velegrakis. Fine-grained controversy detection in wikipedia. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1573–1584, 2015.

[9] T. H. Cao, T. M. Tang, and C. K. Chau. *Data Mining: Foundations and Intelligent Paradigms*, volume 23, chapter Text Clustering with Named Entities: A model, experimentation and Realization, pages 267–287. Springer, Year.

[10] A. Caputo, P. Basile, and G. Semerato. Integrating named entities in a semantic search engine. In *Proceedings of the 1st Italian Information Retrieval Workshop*, 2010.

[11] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, 2004.

[12] J. Guo, G. Xu, X. Cheng, and H. Li. Named entity recognition in query. In *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2009, Boston, MA, USA, July 19-23, 2009*, pages 267–274, 2009.

[13] S. Hensman. Construction of conceptual graph representation of texts. *HLT-SRWS '04 Proceedings of the Student Research Workshop at HLT-NAACL*, 2004.

[14] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 670–681, 2002.

[15] E. Ioannou, W. Nejdl, C. Niedere, and Y. Velegrakis. On-the-fly entity-aware query processing in the presence of linkage. *Proceedings of the VLDB Endowment*, 2010.

[16] N. M.-F. J. Leskovec, M. Grobelnik. Learning sub-structures of document semantic graphs for document summarization. In *Workshop on Link Analysis and Group Detection (LinkKDD)*, 2004.

[17] Lucene. https://lucene.apache.org.

[18] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 115–126. ACM, 2007.

[19] R. Mihalcea and D. Moldovan. Document indexing using named entities, 2001.

[20] R. Mihalcea and D. I. Moldovan. Document indexing using named entities. In *In Studies in Informatics and Control*, 2001.

[21] T. Roelleke and J. Wang. Tf-idf uncovered: a study of theories and probabilities. *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, 2008.

[22] B. Steven, E. Loper, and E. Klein. Natural language processing with python. *O'Reilly Media Inc.*, 2009.

[23] S. Tata and G. M. Lohman. SQAK: doing more with keywords. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 889–902. ACM, 2008.

[24] The OpenCalais System. http://www.opencalais.com.

[25] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 2008.

[26] L. Zhang and Y. Yu. Learning to generate cgs from domain specific sentences. *Lecture notes in Computer Science*, pages 44–57, 2001.