Natural Language Querying System Through Entity Enrichment

Joshua Amavi, Mirian Halfeld Ferrari, Nicolas Hiot

Natural Language Querying System through Entity Enrichment

Joshua Amavi[1], Mirian Halfeld-Ferrari[2], and Nicolas Hiot[1,2]

[1] EnnovLabs, Paris, France (https://en.ennov.com/)
[2] Université d'Orléans, INSA CVL, LIFO EA, Orléans, France
{jamavi,nhiot}@ennov.com {mirian,nicolas.hiot}@univ-orleans.fr

Abstract. This paper focuses on a domain expert querying system over databases. It presents a solution designed for a French enterprise interested in offering a natural language interface for its clients. The approach, based on entity enrichment, aims at translating natural language queries into database queries. In this paper, the database is treated through a logical paradigm, suggesting the adaptability of our approach to different database models. The good precision of our method is shown through some preliminary experiments.

Keywords: NLI · NLP · Database query · Question answering.

1 Introduction

Graph database querying systems adapted to *domain experts*, and not only to database experts, deserve great attention nowadays and become an important research topic. Query language such as SPARQL or CYPHER are powerful tools but require knowledge of the database structure in order to retrieve information. To simplify the accessibility of such databases, the research of natural language interface (NLI) to (structured) databases receives considerable attention today [15, 18]. The idea of NLI is to allow users to focus on the semantic of what they want rather than on how to retrieve it.

This paper describes a practical solution for simple natural language queries on an RDF database, developed for clients of Ennov, a French enterprise specialised in building software solutions to the life sciences industry. Our proposal focuses on the enterprise needs, *i.e.*, factoid queries concerning instances of one RDF 'class', but achieves good results allowing (i) to envisage its use to other domains and (ii) to extend its ideas to more complex queries. The proposal consists in translating a given natural language query (denoted as NL-query) in a database query (denoted as DB-query). In this paper, we use a logical formalism to express database and DB-queries which can be easily translated to any graph, or relational model (and thus to queries on SQL, SPARQL, etc).

In this context, the main contributions of this paper are:

– An original method, based on entities' enrichment, for translating a NL-query into a DB-query. Indeed, entity extraction is a subtask of NLP (Natural Language Processing) and consists of identifying part of an unstructured

text that represents a named entity. After identifying entities connected to a specific domain, a classification into different entity types is possible. Following this classification, some of them are merged and a set of enriched entities is obtained. DB-queries are built from this set of enriched entities.
- An approach composed by two distinct phases: a domain specific pre-processing step and a general query-generating step. The pre-processing step puts in place the general environment which guides query translation: lexicons are built (partially) from the information stored in the database, grammars and ontology mappings are set up. Query-generating algorithms classify and enrich extracted entities and then transform the obtained set of enriched entities into database queries.
- A good-precision querying system. Our approach focus on restricted and specialized domain queries which imply a relatively small vocabulary (mostly composed by people and technical terms appearing in the database instance). Our method takes advantages of this context and gives priority to the use of grammar- and lexicon- based tools. The result is an efficient and precise query translation system.
- An approach proposing a non disambiguation of the natural language queries. Indeed, instead of resolving the ambiguity problem intrinsic to natural language, we adopt a lazy approach and consider all possible interpretations, generating all possible database queries. This option avoids the expensive disambiguation process and speed up the whole performance. The same idea is used to solve ambiguity coming from the use of coordinating conjunctions.

Our querying system is available over an RDF database storing information about medical documents. The system translates a NL-query into a DB-query offering a user-friendly interface. Let us briefly introduce each of these queries together with a running example.

NL-query. Ennov's motivation is to offer a querying system capable to allow its users to perform the so-called *facet search*, narrowing down search results by applying multiple filters. Accepted queries are those requiring information on instances of one unique RDF class (denoted here as solar-class). An allowed query selects only the solar-class instances (the nodes of the given type) via properties having the solar-class as domain or range (the in- or out-edges). For instance, supposing that *Book* is a solar-class, a query requiring book instances edited by doctor Alice on cardiology after year 2018 is an allowed query. On the contrary, a query requiring book instances edited by doctor Alice who is cardiologist, is not allowed since *is cardiologist* is not a property (and edge) of the current solar-class. If the user wants to identify doctors who are writers, then he has, firstly, to change his solar-class specification. In other terms, our query are simple queries identifying instances of one class (even if it can also renders values concerning properties of these instances). The NL-query follows the format *Find books which...* (establishing Book as the solar-class), *Find doctors who...* (Doctors as the solar-class), etc.

Now, as a running example, we introduce query Q_{run}. We use it in the rest of the paper to show, step by step, how to obtain a DB-query. When talking specifically about the NL-query version we can write NLQ_{run}.

Find books with title 'Principles of Medicine' co-authored by Bob and Alice and whose price is less than 30 dollars.

DB-query. We use a logical paradigm to express RDF databases and queries. We write *Book(Anatomy)* to express that *Anatomy* is an instance of class *Book* and *writtenBy(Anatomy, Bob)* to express that *Anatomy* has value *Bob* for property *writtenBy*.

We briefly introduce this logic formalism (refer to [4] for some background on this aspect). An *atom* has one of the forms: (*i*) $P(t_1, ..., t_n)$, where P is an n-ary predicate and $t_1, ..., t_n$ are terms (terms are constants or variables); (*ii*) $\top$ (true) or $\bot$ (false); (*iii*) (t_1 *op* α_2), where t_1 is a term, α_2 is a term or a character string, and *op* is a comparison operator. A *fact* is an atom $P(u)$ where u has only constants. A *database schema* is a set of predicates $\mathbb{G}$ and a *database instance* is a set of facts (denoted by $\mathbb{D}$) on $\mathbb{G}$.

A *(conjunctive) query* q over a given schema has the rule-form $R_0(u_0) \leftarrow R_1(u_1) \dots R_n(u_n), comp_1, \dots, comp_m$ where $n \geq 0$, R_i $(0 \leq i \leq n)$ are predicate names, u_i are tuples of terms of appropriate arity and $comp_j$ $(0 \leq j \leq m)$ are comparison formulas involving variables appearing in at least one tuple from u_1 to u_n. We denote $head(q)$ (respect. $body(q)$) the expression on the left hand-side (respect. right hand-side) of q . The answers for q are tuples t only with constants. For each t there exists a mapping h_t (which maps variables to constants and a constant to itself) such that $\{R_1(h_t((u_1)), \dots, R_n(h_t((u_n))\} \subseteq \mathbb{D}$, the conjunction of all $h_t(comp_j)$ is evaluated to true (according to the usual semantic of opertors *op*) and $h_t(u_0) = t$. In this rule-based formalism, the union is expressed by allowing more than one rule with the same head. For instance, $q(X) \leftarrow writtenBy(X, Bob)$ together with $q(X) \leftarrow editedBy(X, Bob)$ express a query looking for documents written or edited by *Bob*.

Book is the solar-class in NLQ_{run} and thus the DB-query should return the identifiers of book instances. The following conjunctive DB-query is the DBQ_{run} – it includes all the conditions imposed on the books being selected.

$Q(x) \leftarrow Book(x), hasTitle(x, y_1), writtenBy(x, y_2), Person(y_2), writtenBy(x, y_3),$
$Person(y_3), hasPrice(x, y_4), (y_1 =' Principles\ of\ Medicine'),$
$(y_2 = :\text{bob}), (y_3 = :\text{alice}), (y_4 < 30).$ □

Paper Organization. Our approach is depicted in Sections 2 and 3 while implementation and testing results are presented in Section 4. Section 5 concludes the paper with some related work and perspectives.

2 Entity Extraction and Enrichment

In this paper we define a *simple entity* as the tuple $E = (\mathcal{V}, \mathcal{T}, m)$ where $\mathcal{V}$ and $\mathcal{T}$ are lists containing values and lexical types, respectively, and m is a mapping such that $\forall v \in \mathcal{V}, \exists T \subseteq \mathcal{T}, m(v) \rightarrow T$.

Indeed, during the entity extraction phase, ambiguity, an inherent problem in many steps of natural language processing, exists: it concerns the type (*e.g.Paris* can refer to a city or a person) or the value (*e.g.*, several people in the database have the same name). Generally, we seek to eliminate this ambiguity by keeping only the most likely solution. Such a solution may introduce contradiction *w.r.t.* the text semantics. In our approach we explicitly reveal ambiguity (a value may be associated to different types) and we keep track of multiple interpretations during this extraction step, a convenient solution for querying, if we consider the situations where ambiguity can be represented by an *OR* connective.

Entity Extraction (EE) or Named Entity Recognition (NER) is a subtask of NLP and consists of identifying part of an unstructured text that represents a named entity. This task can be performed either by grammar-based techniques or by a statistical models such machine learning (refer to [11] for a complete introduction in the domain). Statistical approaches are widely used in the industry because they offer good results with the latest research and the work of giants like Google, Facebook or IBM. However, these approaches mainly require a lot of data to get good results, implying high costs. More conventional grammar-based methods are very useful for dealing with small data sets.

Entity extraction. Our proposal consists in applying different grammar- or lexicon-based methods together in order to extract simple entities from a given NL-query. The combination of their results allow us to improve entity extraction. The initial parsing step is followed by two different entity extraction methods. One consists of looking up on dictionaries (lexicons) for qualifying an entity. The other is based on local grammars. Notice that the dependency tree resulting from the parsing phase may be used for guiding entity extraction with local grammars.

Parsing. In our approach, tokenization (*i.e.*, determine the words and punctuation), part-of-speech (POS) tagging (*i.e.*, determine the role of the word in a sentence), lemmatization (*i.e.*, determine word canonical form), stematization (*i.e.*, strip word suffixes and prefixes) and dependence analysis are achieved by SpaCy [1] built on a convolutional neural network (CNN) [9] learned from a generic English corpus [16]. The dependency tree produced by SpaCy [8] guides different choices in some of the following steps of our approach. Fig 1 illustrates part of the dependency tree for NLQ_{run}.

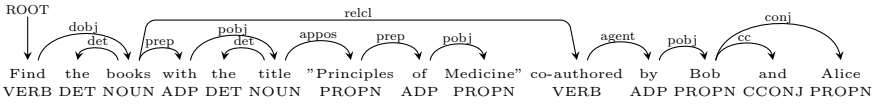

Fig. 1: Dependency tree and POS tagging

Lexicons. When, as in our case, we deal with entity extraction founded on a relatively small database, one could envisage to verify, for each value in the database, whether it appears in the text. However, to improve performance and ensure low coupling, our option consists in building lexicons (*i.e.*, lookup tables) from the database and then in using them as dictionaries containing a summary of the database instance.

LEXICONS

EntityName	*Lexemes*
:alice	*Alice, Wonderful, Wonderful Alice*
:bob	*Sponge, Bob, Sponge Bob*
...	
author	*co-authored, written by, created by*
...	
lt	*less than*
...	
...	

INVERSE INDEX

Pointer to	Index Entry (word from text)
:alice	*Alice*
:alice	*Alice Wonderful*
...	
:bob	*Bob*
author	*co-authored*
lt	*less than*
:alice	*Wonderful*
...	

Ontology-Mappings

EntityName	LexType	DBType
:alice	Person	Person
author	Context	Author
:bob	Person	Person
title	Context	Title
:alice	Doctor	MedicalDoc

DBType	Predicate
Person	Person (X)
Author	writtenBy (_, X)
Book	Book (X)
Title	hasTitle (_, X)
MedicalDoc	Doctor (X)

Operator Dictionary

EntityName	Comparator
lt	$<$
leq	$\leq$
gt	$>$
geq	$\geq$
...	...

Fig. 2: Auxiliary structures built in the pre-processing phase

Fig. 2 illustrates an inverse index pointing to lexicons. It helps finding *EntityValue* quickly. For instance, if *Wonderful Alice* is the string found in the text, the index allows us to find how to refer to it, *i.e.*, with its *EntityValue*, :alice. With *EntityValue* we have access to information stored in auxiliary structures denoted here by Ontology-Mappings. In order to simplify notations, we consider the existence of three functions that render information over Ontology-Mappings. They are: (1) lexT: given *EntityValue*, the function renders its lexical type indicating entity's lexical role, detected during the lexical analysis; (2) dbT: given *EntityValue*, the function renders its database type which indicates the semantic associated to the entity in our RDF database and (3) getPred: given the database type, the function renders the associated predicate which is the one we should use in the DB-query. Moreover, the arity of predicates is indicated together with the argument position reserved for an entity having this database type. According to the example in Fig. 2, we have: $\mathsf{lexT}(:alice) = Person$, $\mathsf{dbT}(:alice) = Person$ and $\mathsf{getPred}(Person) = Person(X_{person})$. Similarly, lexeme *co-authored by* refers to the entity value *author* and we obtain $\mathsf{lexT}(author) = Context$, $\mathsf{dbT}(author) = Author$ and $\mathsf{getPred}(Author) = writtenBy(X, Y_{author})$.

Local grammar. There is no sense in storing possible values for attributes associated to huge domains. Dates, general numerical attributes (*e.g.*, prices, weight, etc) or even publication or section titles are not stored in our database. Entity extraction, for such cases, is based on local grammars. Currently we have designed 15 local grammars as a support for our entity extraction mechanism.

Entity	*EntityValue*	lexT
E_1	book	Context
E_2	title	Context
E_3	author	Context
E_4	:bob	Person
E_5	:alice	Person
E_6	price	Context
E_7	lt	Operator
E_8	'Princ. of Medecine'	Text
E_9	30	Number

Fig. 3: Simple entities extracted from NLQ_{run}

Example 1. We show our NLQ_{run} with expressions in boxes indicating the entities detected after the extraction step. Fig. 3 summarizes obtained entities: E_1-E_7 via Lexicons while E_8-E_9 are detected by local grammars. Fig. 3 does not represent set notation to avoid figure overload. Here, each entity has only one value and each value is associate to a singleton.

Find the books$_1$ with the title$_2$ "Principles of Medicine"$_8$ co-authored by$_3$ Bob$_4$ and Alice$_5$ and whose price$_6$ is less than$_7$ 30$_9$ dollars. □

Once we have extracted simple entities, we classify them into categories in order to decide about their fusion or not. Our goal is to build enriched entities which concentrate information initially available in the detected simple entities. An enriched entity is a first-class citizen which will guide the construction of DB-queries while simple entities are auxiliary ones considered as second-class citizens and committed to integrate the enriched entity.

Entity Enrichment. An *enriched entity* is a relation. It can thus be seen as table (as in the relational model) with schema $E_e[EntityValue, DBType, LexType, op]$. The entity E_e itself is a relation instance, *i.e.*, a set of functions (tuples) associating each attribute in the schema to a value. Thus, each tuple maps: (i) $EntityValue$ to the value v of an entity as represented in a Lexicon (Fig 3), (ii) $DBType$ and $LexType$ to $\mathsf{dbT}(v)$ and $\mathsf{lexT}(v)$, respectively (Fig 2), and (iii) *op* to a comparison operation indicating the kind of comparison imposed on the entity value. By default, the comparison operation *op* is *equal to*.

On the other hand, we distinguish the following classes of simple entities:

• A *reference entity* is the one chosen to evolve, *i.e.*, to be transformed into an enriched entity. It corresponds to an instance value in a database (*e.g.*, people names, document titles, dates, etc) .

• An *operator entity* E_{cp} represents words which indicate that another (reference) entity $E = (\mathcal{V}, \mathcal{T}, m)$ is constrained by a comparison condition. A connection between E and E_{cp} in the dependency tree determines E as the reference entity. Then, E evolves to an enriched entity E_e such that for each $v \in \mathcal{V}$ we have $(v, \mathsf{dbT}(v), \mathsf{lexT}(v), op)$ in E_e where *op* is defined by E_{cp}, according to an available dictionary (see example in Fig. 2). Notice also that *op* corresponds to the operator used in the DB-query (Section 1) where *comp* atoms have the general format $(t_1\ op\ \alpha_2)$. In Example 1, expression *less than* is an operator entity having 30 as its reference entity.

• A *context entity* E_C gives information about the type of another (reference) entity. Once again, the dependency tree obtained during the parsing, determines the reference entity E which evolves to a new enriched entity E_e according to Algorithm 1. In Example 1, *price* is a context entity and 30 as its reference. Similarly, *title* is a context entity and *Principles of Medicine* its reference (Fig 1).

We remark that the first For-loop of Algorithm 1 transforms a simple entity into an enriched one. Given a simple entity E, $extend(E)$ is the relation instance obtained by converting E into its extended counterpart. Notice that the entity evolution process starts with a simple entity which becomes an enriched entity, but such an enriched entity can continue evolving.

Example 2. From Fig. 3 and our auxiliary structures (partially depicted in Fig. 2) we obtain the following enriched entities:
$E_{e0} = \{(book, Book, Context, =), \};$
$E_{e1} = \{('Princ.\ of\ Medicine', Text, Text, =),$
$('Princ.\ of\ Medicine', Title, Context, =)\}$

Algorithm 1: contextEnrichment

Input: $E_C = (\mathcal{V}_C, \mathcal{T}_C, m_C)$ and $E = (\mathcal{V}, \mathcal{T}, m)$
Output: E_e: an instance over schema $E_e[EntityValue, DBType, LexType, op]$

```
1: for all v ∈ V do
2:     Insert (v, dbT(v), lexT(v), op) in E_e;
3:     for all u ∈ V_C do
4:         for all v ∈ V do
5:             Insert (v, dbT(u), lexT(u), op) in E_e;
```

$E_{e2} = \{(:\text{bob}, Person, Person, =), (:\text{bob}, Author, Context, =)\}$
$E_{e3} = \{(:\text{alice}, Person, Person, =), (:\text{alice}, Author, Context, =)\}$
$E_{e4} = \{(30, Number, Number, <), (30, Price, Context, <)\}$.

All entities are enriched ones. E_{e1} results from the integration of context entity E_2 into E_8, E_{e2} results from integration of E_3 into E_4 and E_{e3} results from integration of E_3 into E_5. Notice that coordinating conjunction *and* in NLQ_{run} implies the existence of these two latter independent enriched entities. E_{e4} results from the integration of operator entity E_7 to E_9. Entity E_{e0} is just the enriched version of E_1. It corresponds to the solar-class. □

Now, a NL-query may include multiple conditions (or filters) connected by coordinating conjunctions. Our current version deals only with *and* and *or*, even if we intend to extend this initial proposal to more complex coordinating conjunctions such as *nor*, *for*, etc . Coordinating conjunctions are expressed through logical formulas which guide the construction of the DB-query, by specifying: (i) the kind of atoms *comp* it will have and (ii) whether the query is defined by one of several rules. Taking into account coordinating conjunctions implies entity enrichment. Let $E_1 = (\mathcal{T}_1, \mathcal{V}_1, m_1)$ and $E_2 = (\mathcal{T}_2, \mathcal{V}_2, m_2)$ be two simple entities having the same LexType. If, in the query text, these two entities are connected by an *or*, they are merged, forming a new enriched entity composed by $extend(E_1) \cup extend(E_2)$. The original entities do not exist any more. Otherwise, if in the text, these two entities are connected by an *and*, they are kept as independent ones.

Example 3. In Example 2, $E_{e2} = \{(:\text{bob}, Person, Person, =), (:\text{bob}, Author, Context, =)\}$ and $E_{e3} = \{(:\text{alice}, Person, Person, =), (:\text{alice}, Author, Context, =)\}$ are independent entities. When considering NLQ_{run} these entities do not merge because the coordinating conjunction is an *and*. If we change the sentence to '*written by Bob or Alice*', entities are merged resulting in:

$E_{enew} = \{(:\text{bob}, Person, Person, =), (:\text{bob}, Author, Context, =),$
$(:\text{alice}, Person, Person, =), (:\text{alice}, Author, Context, =)\}$. □

Queries may have multiple coordinating conjunctions as illustrate in sentence *written by Alice or Bob and Charlie* and, in this case, its interpretation (due to the ambiguity of natural language) can vary, resulting in the logic formula $\mathcal{F}_1 \equiv (X = :\text{alice}) \vee ((X = :\text{bob}) \wedge (X = :\text{charlie}))$ or in the formula $\mathcal{F}_2 \equiv ((X = :\text{alice}) \vee (X = :\text{bob})) \wedge (X = :\text{charlie})$. To avoid erroneous query answers,

one may envisage to take into account all the alternative interpretations, or to give choices to the user. In our approach we do not plan interactions with the user and thus, we propose to consider a larger interpretation, *i.e.*, to overcome ambiguity by replacing a mixed *and-or* sentence by an only-*or* sentence. Thus, let $\mathcal{F}$ be the logic formula obtained from a sentence with coordinate conjunctions. If $\mathcal{F}$ contains both $\vee$ and $\wedge$, then we replace it by the formula $\mathcal{F}'$ composed only of $\vee$. The idea is based on the fact that any answer satisfying $\mathcal{F}$ also satisfies $\mathcal{F}'$. In that way, when multiple coordinate conjunctions are present, the DB-query will be represented by multiple rules with the same head.

3 Building DB-Queries from Enriched Entities

Once our NL-query is analysed and all enriched entities are completed, the DB-query is generated by Algorithm 2. The algorithm starts by considering entity E_{e0} (line 3) which has a special role since it specifies the solar-class, *i.e.*, the class on which the query focuses. The query is initialized with a body composed by one unique atom over the predicate associated to the solar-class. Notice the use of function bAt which is responsible for building an atom for the query being constructed. The predicate symbol to be used in the construction of an atom is found via the value of attribute $DBType$ in E_{e0} – which is then used as an input for function getPred. In Example 2, *Book* is the value of attribute $DBType$ in E_{e0} and the name of the associated unary predicate. Atom A_0 in our case is $Book(x)$. Notice that Algorithm 2 builds only queries whose answers are books' identifiers (*i.e.*, instantiations of x). Our initial query is thus $q(x) \leftarrow Book(x)$.

Lines 11 to 15 of Algorithm 2 consider entities E_e enriched with a context entity. If in E_e there are more than one tuple t for which the value of attribute $lexType$ is "Context", then E_e is an entity obtained after taking into account coordinating conjunction *or*. Each tuple t having value "Context" for attribute $lexType$ has to be grouped together with the tuple t' representing its reference. On line 14, Algorithm 2 groups each t with another $t' \in E_e$ having the same value for attribute $EntityValue$. From information in t and t' we build a list l, added to set *Parts*. Each $l \in Parts$ is a list of atoms to be added to the body of the query under construction. Notice that Algorithm 2 divides E_e's tuples into parts (or lists). Each list in *Parts* generate a new distinct query with the same *head*. Indeed, on line 24, in the for-loop, each list l is used to create a new query q' – continuing the construction of a query q already in $\mathcal{Q}$. If there are more than one list l in *Parts*, there will be more than one query q'.

From Example 2, E_{e2} has two tuples. Let t_1 be the first tuple for which $\mathsf{getPred}(t_1(DBType)) = \mathsf{getPred}(Person)) = Person$ and t_2 be the second one for which $\mathsf{getPred}(t_2(DBType)) = \mathsf{getPred}(Author)) = writtenBy$. The function bAt can be used to build atoms that will be added to $body(q)$. Notice that bAt also takes into account information concerning positions marked as the place for the entity value in the atom being built. Thus, the new variable y representing the entity is placed accordingly. In binary predicates x is always the other variable. Atoms *comp* may associate a value to variable y. Thus, on line 14 list $\langle Person(y_2), writtenBy(x, y_2), (y_2 = \text{:bob})\rangle$ is added to *Parts* and on line 24 the

Algorithm 2: EntitiesToQueries

Input: $\mathcal{E}$ an enriched entity set $\{E_{e0}, E_{e1}, \dots\}$
Output: $\mathcal{Q}$ a set of query rules, *i.e.*, the DB-query with one or more rules

1: $\mathcal{Q} := \emptyset$
2: **for all** enriched entity E in $\mathcal{E}$ **do**
3: **if** E is E_{e0} **then**
4: $\{(eval, dbTval, lTval, opval)\} := E$
5: $A_0 := \mathsf{bAt}(dbTval, x)$ // Build the first atom for the query's body
6: $\mathcal{Q} := \{q(x) \leftarrow A_0\}$
7: **else**
8: $Parts := \emptyset$ // Set of list of atoms. Each $l \in Parts$ is a list of atoms whose conjunction should be added to query's body.
9: $E' := \emptyset$ // Set storing E's tuples already treated
10: // Treatment of entities enriched with a context
11: **for all** tuple $t = (eval, dbTval, \text{"Context"}, opval)$ in E **do**
12: Let $t' \in E$, s.t $t' = (eval, dbTval', lTval', =)$ and $lTval' \neq$ "Context"
13: $y := GetNewVar()$
14: $Parts := Parts \cup \{\langle \mathsf{bAt}(dbTval', y), \mathsf{bAt}(dbTval, y), \mathsf{bAtOp}(eval, y, opval)\rangle\}$
15: $E' := E' \cup \{t', t\}$
16: // Treatment of enriched entities without tuples where $lTval =$ "Context"
17: **for all** tuple $t = (eval, dbTval, lTval, opval)$ in $(E \setminus E')$ **do**
18: $y := GetNewVar()$
19: $Parts := Parts \cup \{\langle \mathsf{bAt}(dbTval, y), \mathsf{bAtOp}(eval, y, opval)\rangle\}$
20:
21: $\mathcal{Q}' := \emptyset$
22: **for all** query $q \in \mathcal{Q}$ **do**
23: **for all** list $l \in Parts$ **do**
24: $q' = BuildNewQuery(q, l)$
25: $\mathcal{Q}' := \mathcal{Q}' \cup \{q'\}$
26: $\mathcal{Q} := \mathcal{Q}'$
27: **return** $\mathcal{Q}$

query being built is $q(x) \leftarrow Book(x), Person(y_2), writtenBy(x, y_2), (y_2 = \text{:bob})$. The result obtained with entity E_{e3} is similar. However, entities E_{e1} and E_{e4} are treated in a different way since their lexical types are *Text* and *Number*, respectively. These entities are treated on lines 17-19. For instance, E_{e1} gives rise to list $\langle hasTitle(x, y_1), (y_1 = \text{"}Princ.\ of\ Medicine\text{"})\rangle$. After considering all entities, Algorithm 2 returns set $\mathcal{Q}$ with the following DB-query:

$$\begin{aligned} q(x) \leftarrow & Book(x), hasTitle(x, y_1), Person(y_2), \\ & writtenBy(x, y_2), Person(y_3), writtenBy(x, y_3), asPrice(x, y_4), \\ & (y_1 = \text{"}Princ.\ of\ Medicine\text{"}), (y_2 = \text{:bob}), (y_3 = \text{:alice}), (y_4 < 30) \end{aligned}$$

However, if we consider E_{enew} of Example 3, Algorithm 2 (lines 11 to 15) produces two lists from the same entity, namely, $l_1 = \langle Person(y_2), writtenBy(x, y_2), (y_2 = \text{:bob})\rangle$ and $l_2 = \langle Person(y_3), writtenBy(x, y_3), (y_3 = \text{:alice})\rangle$. Then, on

line 24, each list is considered separately and the query q is replaced by two new queries. At the end, $\mathcal{Q}$ returns a DB-query composed by two rules:

$$q(x) \leftarrow Book(x), hasTitle(x, y_1), Person(y_2), (y_1 = "Princ.\ of\ Medicine"),$$
$$writtenBy(x, y_2), hasPrice(x, y_4), (y_2 = \text{:bob}), (y_4 < 30)$$
$$q(x) \leftarrow Book(x), hasTitle(x, y_1), Person(y_3), (y_1 = "Princ.\ of\ Medicine"),$$
$$writtenBy(x, y_3), hasPrice(x, y_4), (y_3 = \text{:alice}), (y_4 < 30)$$

Finally, consider $E_{enew2} = \{(\text{:bob}, Person, Person, =), (\text{:bob}, Author, Context, =), (\text{:bob}, Editor, Context, =)\}$. The resulting lists on line 14 are $l1 = \langle Person(y_2), writtenBy(x, y_2), (y_2 = \text{:bob})\rangle$ and $l_2 = \langle Person(y_3), editedBy(x, y_3), (y_3 = \text{:bob})\rangle$ and $\mathcal{Q}$ also returns a DB-query composed by two rules. Here we are looking for *books edited or written by Bob*.

Thus, queries can be directly generated from enriched entities. Currently we only deal with conjunctive queries – easily translated to SQL or SPARQL.

4 Implementation and Experimental Results

In order to validate our system, we implemented it in the form of a pipeline which allows us to divide the separate stages and explore various combinations. For lexicon-based entity extraction, Apache SolR [2] is used with its text tagger, an inverted index and n-gram algorithm [12]. It allows lexemes detection even with typographic errors. We also use a combination of hand-written grammars together with a Facebook project called Duckling [3] which provides powerful tools for extracting entities such as numbers or dates. Each extraction step is performed independently and simple entities are defined by taking into account all different methods. In particular, if several approaches identify entities in the same place (but not necessarily with the same bounds), we keep only the entity resulting from the union of the overlapping entities to represent the ambiguity. To implement this pipeline and link each step, we use the RASA NLU framework [6] in combination with SpaCy for the parsing phase. Notice that the preprocessing part is based on generic grammars and lexicons. Some lexicons are generated automatically by considering values in the RDF database (*e.g.*, first and last name for *Person*). Hand-written lexicons such as those for operators and contexts concerning dates (*e.g.*, application, archive, creation, expiration) or persons (*e.g.*, author, signatory) are also used. Partial matches are managed using multiple lexemes when possible (*e.g. create by*, *create with*, *create*, ...).

We conduct our preliminary experiments on an RDF database concerning medical publications. The database has 66 classes (possible candidates for a query solar-class) with a total of 29327 class instances. In our tests, about 10 classes have been used as solar-classes. These tests have considered entity extraction and enrichment phases. Ambiguity has not been tested and thus we only take into account one value per entity. The evaluation is done by analysing the obtained enriched entities. Table 4 shows the results of our experiments on a set of 113 NL-queries (varying number of *and* and *or*). It summarizes the precision, recall, f1-score, and the weighted mean (weighted by support) obtained for each

dbT on the NL-queries set. As our system is implemented as a pipeline, we intend to perform tests step by step, in order to identify the impact of each step.

Our precision is good, indicating that most of our detected entities are the expected ones. This is clearly a consequence of the effective use of lexicons and grammars. For recall, our results are not bad, but weaker than our precision, indicating that some entities are not detected. Lower precision on some dbT like *creation_date* or *doc_author* is partially explained because ambiguity is not taken into account in our experiment, but entities giving rise to these types are enriched with a similar context and associated with both types. A similar issue occurs with the recall for *department* and *doc_unit*. In our test database they are semantically close. So, we have significant overlap on the two lexicons (a lot of lexemes are shared, adding ambiguity to the type). Our current work consists in improving lexicons for context detection, in particular those generated automatically.

DBType	precision	recall	f1-score	support
solar_class	1.00	0.62	0.77	82
application_date	1.00	0.62	0.76	13
archive_date	0.50	0.67	0.57	3
creation_date	0.60	0.60	0.60	5
expiration_date	1.00	0.50	0.67	2
customers	1.00	0.60	0.75	5
department	1.00	0.11	0.20	9
sector	1.00	0.95	0.98	21
doc_author	0.77	0.63	0.70	38
doc_signatory	0.90	0.86	0.88	21
doc_status	1.00	0.50	0.67	6
doc_unit	1.00	0.27	0.43	11
...	...	...	...	...
Weighted avg.	0.86	0.59	0.67	295

Fig. 4: Results on enriched entities

5 Related Work and Concluding Remarks

Recently, NLI has been widely discussed in the literature. Some work focuses on augmenting the expression power of queries while others on domain-independence. For instance, in [18] authors propose the use of binary templates rather than semantic parses to better understand complex queries while [15] proposes a cross-domain NLI with a general propose question tagging strategy. Several work (such as in [19, 5, 7]) consider RDF question/answering (QA). Aggregate queries are considered in [10]: the authors propose a method to automatically identify the aggregation and transform it into a SPARQL aggregate statement. Methods used vary a lot. In [13, 17, 18] authors base their approach on NLP techniques with entity extraction and grammars, while in [14, 15] they use neural networks.

The paper presents a method where enriched entities allow us to translate NL-queries into DB-queries. The use of a logical paradigm to deal with databases shows that our method can be adapted to different data models. Our approach is divided into a domain-dependent pre-processing and domain-independent query generation phases. The first step, responsible for building lexicons, grammar-tools and ontology mappings, also sets up general propose tools which can be considered as domain-independent (*e.g.*, grammars for date recognition). The second step of our method can be applied on any domain, provided the ontology mappings are set up. This division allows us to deal with possible extensions and improvements separately. We are currently considering extensions of Algorithm 2 in order to deal with queries on more than one solar-class or aggregate queries. We also plan to extend entity extraction by including alternative approaches such as machine learning to complete grammars (*e.g.* for title identification).

References

1. https://spacy.io
2. https://lucene.apache.org/solr/
3. https://github.com/facebook/duckling
4. Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases, vol. 8. Addison-Wesley Reading (1995)
5. Amsterdamer, Y., Kukliansky, A., Milo, T.: A natural language interface for querying general and individual knowledge. Proc. VLDB Endow. **8**(12), 1430–1441 (2015)
6. Bocklisch, T., Faulkner, J., Pawlowski, N., Nichol, A.: Rasa: Open Source Language Understanding and Dialogue Management. arXiv:1712.05181 [cs] (2017)
7. Fader, A., Zettlemoyer, L., Etzioni, O.: Open question answering over curated and extracted knowledge bases. In: The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014. pp. 1156–1165. ACM (2014)
8. Honnibal, M., Johnson, M.: An improved non-monotonic transition system for dependency parsing. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. pp. 1373–1378. Association for Computational Linguistics, Lisbon, Portugal (September 2015)
9. Honnibal, M., Montani, I.: spacy 2: Natural language understanding with bloom embeddings, convolutional neural networks and incremental parsing. To appear **7**(1) (2017)
10. Hu, X., Dang, D., Yao, Y., Ye, L.: Natural language aggregate query over RDF data. Inf. Sci. **454-455**, 363–381 (2018)
11. Jurafsky, D., Martin, J.H.: Speech and Language Processing. Stanford Univ., 3rd draft edn. (2019), https://web.stanford.edu/ jurafsky/slp3/
12. Kim, J.Y., Shawe-Taylor, J.: Fast string matching using an n-gram algorithm. Software: Practice and Experience **24**(1), 79–88 (1994)
13. Steinmetz, N., Arning, A.K., Sattler, K.U.: From Natural Language Questions to SPARQL Queries: A Pattern-based Approach. In: BTW 2019. pp. 289–308 (2019)
14. Utama, P., Weir, N., Basik, F., Binnig, C., Cetintemel, U., Hättasch, B., Ilkhechi, A., Ramaswamy, S., Usta, A.: An End-to-end Neural Natural Language Interface for Databases. arXiv preprint arXiv:1804.00401 (2018)
15. Wang, W.: A cross-domain natural language interface to databases using adversarial text method. In: Database. vol. 1, p. 4 (2019)
16. Weischedel, R., Hovy, E., Marcus, M., Palmer, M., Belvin, R., Pradhan, S., Ramshaw, L., Xue, N.: OntoNotes: A large training corpus for enhanced processing. Handbook of Natural Lang. Processing and Machine Translation p. 59 (2011)
17. Zafar, H., Napolitano, G., Lehmann, J.: Formal Query Generation for Question Answering over Knowledge Bases. In: European Semantic Web Conference. pp. 714–728. Lecture Notes in Computer Science (2018)
18. Zheng, W., Yu, J.X., Zou, L., Cheng, H.: Question answering over knowledge graphs: question understanding via template decomposition. Proceedings of the VLDB Endowment **11**, 1373–1386 (2018)
19. Zou, L., Huang, R., Wang, H., Yu, J.X., He, W., Zhao, D.: Natural language question answering over RDF: a graph data driven approach. In: International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014. pp. 313–324. ACM (2014)