# Towards an Ontology-Driven Evolutionary Programming-Based Approach for Answering Natural Language Queries against RDF Data

Sebastian Schrage
Georg-August University of Göttingen, Institute of Computer Science
schrage@cs.uni-goettingen.de

Wolfgang May
Georg-August University of Göttingen, Institute of Computer Science
may@cs.uni-goettingen.de

## ABSTRACT

In this paper, we present an ontology-driven evolutionary learning system for natural language querying of complex relational databases or RDF graphs to give users who are not familiar with formal database query languages the opportunity to express complex queries against a database. This approach learns how to arrange and when to use given functions to process Natural Language Queries (NLQ).

## 1. INTRODUCTION

Natural language interfaces for databases (NLIDB) are likely the easiest way for a user to access a database. It does not require the user to learn the specific query language nor the schema or ontology of the data set. But this lack of knowledge must be compensated by the interface. It not only has to understand the user input and to extract the information from the natural language query (NLQ), but also the user could have a different concept in mind than the one implemented. This can range from a smaller deviation in the vocabulary, or using abbreviations or incomplete names, over using ambiguous formulations to using relationships which are not in the model or fusing different entities to one. Therefore, a NLIDB should be flexible enough to allow the user to operate on her concepts, not on those of the implementer.

This approach consists of two major parts, the evolutionary agent framework loosely based on the work of Turk [11] and Hoverd et al. [2] and the NLQ-to-SPARQL application of this framework, which uses pre-processed NLP data by the Stanford's NLP Core [3] and ontology-based methods to translate a NLQ into a SPARQL query against a given RDF database which is described by an ontology. According to the definition from Vikhar [12] the framework can be categorized as evolutionary programming, since other than in genetic algorithms, the structure of the subprograms is fix and only the execution order of those can differ and other than in evolutionary strategies the data types of the solution is not limited to a numeric vector. Due to the nature of the NLQ it can easily be decomposed into several layers of sub-objectives, therefore a multiobjective evolutionary algorithms (MOEAs) approach like extensively investigated from Li et al.[13] was used. This provides the possibility if the system is not able to provide the correct solution, to train it with further example queries of this kind with corresponding SPARQL queries and the framework extends the model via the evolutionary learning algorithm to improve it and to learn this new kind of queries.

## 2. RELATED WORK.

Basically there are two environments in which NLIDB systems are developed, for Knowledge Graphs (KG) like DBpedia[1] and for smaller data sets like Mondial [7]. For KGs with huge amounts of data and entities but no reliable or well-defined ontology given, approaches based on predefined graph pattern matching like the approach from Steinmetz et al. [10] or pattern learning approaches like STF from Hu et al. [9] relying less on ontologies have shown the most success.

On the other hand for smaller data sets with well-defined ontologies, which is also the scope of this work, approaches more focused on ontology usage like Athena from Saha et al. [6] or schemata usage like Precise from Popescu et al. [4] have shown the better results. Both approaches first analyze the NLQ, then assign values to recognized parts representing how confident those parts are considered and then try to connect them to a minimal graph that spans all parts that are considered evident with weighted edges according to the confidence values or with high penalties if not found at all.

**Structure of the paper:** Next, a short overview of the system architecture is given, followed by description of the learning framework. Then, the NLQ-to-SPARQL application is discussed, with a some example queries. If the approach could not answer a question correctly, a brief explanation is given why. Last a brief conclusion is given.

## 3. SYSTEM OVERVIEW

This approach is based on evolutionary programming. The central component is an agent whose input is obtained by preprocessing the NLQ with NLP Core [3] and which outputs a SPARQL query (cf. Figure 1).

The system is initialized with an ontology that covers the application domain. The ontology, given as an OWL ontology, is analyzed by RDF2SQL [5] and the results are stored in the *Semantical Data Dictionary* which is a collection of relational tables stored in an SQL database. When an NLQ is

$31^{st}$ GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 11.06.2019 - 14.06.2019, Saarburg, Germany.

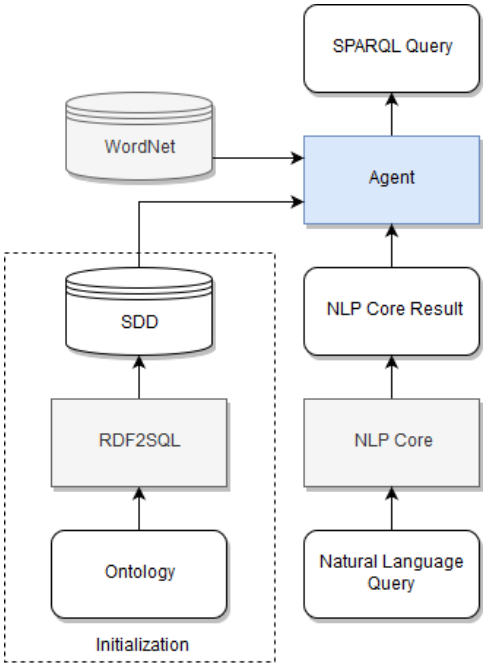


**Figure 1: Overall architecture of the approach**

asked, it is first processed with Core NLP [3] using its part of speech module, the entity recognition module and the grammatical dependencies module. Then the preprocessed NLQ is given to an agent which returns a SPARQL query, which can be stated against an RDF data storage or further processed by ODBA applications.
As depicted in Figure 1, at runtime there is a single agent. During the learning phase, there are multiple agents, and the learning phase results in the "fittest" agent for a given learning set, as described in the following section.

# 4. LEARNING FRAMEWORK

Therefore the structure for agents that are subject to evolutionary programming has been developed accordingly: The inner structure of an agent consists of application-specific *nodes*. There are different node *types*, and from each type there may be multiple instances. The general idea of the node types is to provide a set of operations which might be useful for solving the task, but it is not known, which of them are needed, in which order and in which cases they must be executed and with which settings, to reach the objectives. Additionally, there are connections between nodes for the data flow inside the agent. The information flow is handled in so called *products*, which are just an application-specific predefined encapsulation of arbitrary data types. Which kind of products and how many at the same time are accepted by the node is type specific. An agent is a network of such nodes (for an example see Figure 2) and the computed solution is returned as a set of products.

## 4.1 General Notions

### 4.1.1 Agent Configuration

The *configuration*, i.e. the concrete internal structure of an agent, implements its functionality as the cooperation of the nodes. It is a directed graph (which may contain cycles) consisting of a set of nodes and a set of connections. There are input nodes, a single output node, and inner, processing nodes. The graph must be connected, i.e. no isolated fragments are allowed.

### 4.1.2 Nodes

The nodes themselves have all the same general structure, each node $n$ of type $t$ has at least one input or one output *conduit*, usually one or several of both. The conduits are typed according to which kind of data, called *products* (cf. Section 5.2) they communicate. The product types are organized in a class hierarchy. The input conduits are enumerated as $\mathsf{in}_1$, $\mathsf{in}_2$, ... with types $\mathsf{type}(\mathsf{in}_i)$ ; the output conduits are enumerated as $\mathsf{out}_1$, $\mathsf{out}_2$, ... with $\mathsf{type}(\mathsf{out}_j)$. There might be several input conduits with the same product type. Nodes have one or more output conduits for every product type that it can produce (which in course can be connected to multiple inputs). A node can generates one or more results of one or more product types.

Every node type $t$ implements a certain functionality, which satisfies a certain *signature* wrt. its inputs $\mathsf{in}_1, \ldots, \mathsf{in}_{c(t)}$, and $\mathsf{out}_1, \ldots \mathsf{out}_{d(t)}$ (i.e. $c(t)$ and $d(t)$ are the indegrees and outdegrees of nodetype $t$, resp.),
$f_t(\mathsf{type}(\mathsf{in}_1), \ldots, \mathsf{type}(\mathsf{in}_{c(t)})) \rightarrow (\mathsf{type}(\mathsf{out}_1)^*, \ldots, \mathsf{type}(\mathsf{out}_{d(t)})^*)$,
where the $^*$ means that there might be zero, one, or more elements (e.g., if the node implements a conditional, one out of two outputs will be set, and if a node cannot do something useful with the current inputs, no output might be generated; or if a list is split, the (only) output is fed with the sequence of all its elements). From a practical point of view, the output can also bee seen as a set of elements of arbitrary product types.

Every output conduit can be connected to multiple input conduits, and every input conduit can have multiple incoming connections from output conduits.

The used product types, the concrete functionality ($f_t$ including the number of input and output conduits, and their product types) of the nodetypes depend on the application. The *structure* of the agent as a graph of nodes of these node types and conduits connecting compatible output and input is subject to learning. Usually, it is started with a concrete proposal of a *standard agent* which is then improved during the learning process.

## 4.2 The Evolutionary Process

The evolutionary process controls the evolution of agents in order to improve their competences. It starts with a set of mutated standard agents. Then, in an iterative process, the agents have a chance to change their configuration every time they reproduce. The basic idea from [2] is that each solution to each problem is assigned an amount of energy. An agent gets energy for correct solutions of the problems. With a growing population of agents, the energy is divided by more agents and pressures them to win more energy overall and suppresses unlimited growth in numbers.

### 4.2.1 Stepwise Evolution

The framework is organized as a sequence of runs. There is a fixed training set $T$ provided by the user consisting of test pairs $t = (p_t, sol_t)$ consisting of a problem $p_t$ and a corresponding solution. The solutions, and often also their components, are assigned an initial energy (=value). Initi-

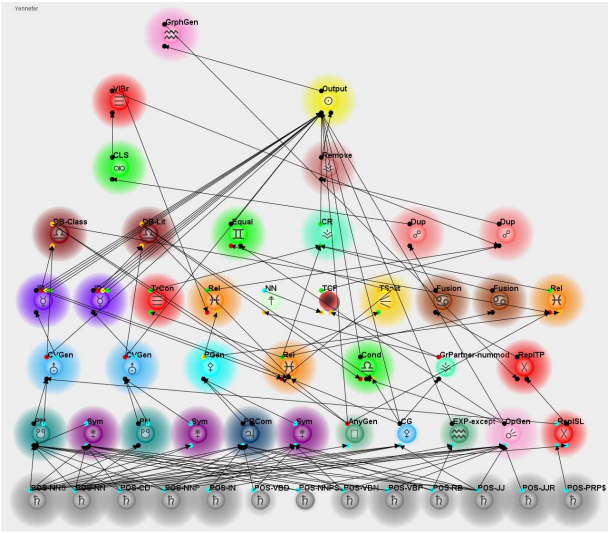


**Figure 2: Visual sketch of a learned agent.** Each node is represented as a circle and each type has a distinct color and icon. The size of the glow around each node represents its activity and the smaller dots represent the conduits of the nodes.

al and new agents can be created from a problem-specific *standard agent*. Each run is done by an agent set, whose population changes by evolution. All agents have to solve the problems, and the produced solutions are evaluated. Then, for each solution (resp. solution component) it is checked to which extent the solution of an agent matches that solution component. The energy assigned to the solution components is distributed to the agents that found it.

Given a threshold $e_{sus}$ that defines how much energy is required to sustain an agent, the next step is to check which agents collected at least $e_{sus}$ energy. Those are then added to the agent set of the next round. If an agent earned more than $2e_{sus}$, it reproduces (i.e., mutates) itself and the offspring is added to the agent set as well. After being unchanged a certain number of runs, it can mutate itself.

#### 4.2.2 Reproduction

If $e \geq 2e_{sus}$ for an agent, it reproduces itself and adds itself and its offspring to the agent set as well. During reproduction one of two scenarios could happen.

1.) The offspring is a perfect copy of its parent, without any changes. This means for the next run there are more agents of this configuration and during reproduction the likelihood of successful mutations is higher.

2.) The offspring is a mutation of the agents, meaning it makes a random number of changes (based on a normal distribution centered around a value $> 0$) on its nodes and connections. These changes can be adding a new node/connection, removing a node/connection or changing the configuration of a node.

## 5. NLQ-TO-SPARQL TRANSLATION

For every application, the specific node types must be designed and implemented. This requires a profound idea of useful small steps of the process. Then, the learning process consists of combining such *local* behavior into a smooth *global* behavior.

For the NLQ-to-SPARQL translation, the task of the agent is to translate the outcome of the Core NLP analysis into a SPARQL query. So, the solution components mentioned above are query fragments. There are different issues to be done and combined by the agent: Named-Entity-Recognition, translation of class names and property names into the notions of the database (represented by its ontology), and the structural generation of a SPARQL query by basic graph patterns (BGPs), logical connectives and conditions, and to deal with the variables.

The training set consists of a set of pairs of NLQs and the corresponding (usually handmade) annotated SPARQL queries that adhere to conventions to give hints to the translation of the sentence structure.

### 5.1 Ontology Representation and Access

The *Semantic Data Dictionary (SDD)* [5] gives comprehensive access to the metadata: Basically it contains the same knowledge as an OWL ontology (where it can be extracted from to provide originally an OBDA RDF-to-SQL mapping), extended with knowledge which *concrete* (sub)classes provide which properties, and their ranges. It is used here instead of the OWL ontology because it is easier to access and does not require further reasoning.

The SDD has no information about the instances in the data set. Since identifying instances in NLQs is one of the major tasks for answering them, a data structure for efficient searching is necessary. Therefore the SDD is extended with an identifier mapping IM: $string \rightarrow (class, property)^*$, e.g. *"Monaco"* $\mapsto ((Country, name), (City, name))$. To identify which properties are potential identifiers for the mapping, the training set is searched for cases where the SPARQL solution contains a variable whose name is not equal to its class – these denote the named entities (Great (Britain), in the example, whose class is Country). For instances of these classes all string-valued properties are searched whether their value equals the name of the training set variable (i.e., "Great Britain"). If so, the property is considered as identifying property and generates an entry for each instance of this class with this property in the identifier mapping.

### 5.2 Products

The products for the NLQ application are divided into two major groups: primitive and compound products. Usually, primitive products can contain complex information, but as a product they are seen as-a-whole (see Table 1 for the exact data composition of each product, where e.g. most products carry the position in the sentence from which they have been derived as information). At the primary stage of the processing, there are primitive products of type nlpdata which is a reduced version of the output of NLP Core [3]. Nlpdata can be turned either into tripleparts or symbols, which are primitive products towards the SPARQL side. Triplepart is an abstract superclass of the product types variable, constant, and predicate, while symbol is the abstract superclass of the product types operator (e.g., $+$, $\leq$, $\geq$, $=$, $\neq$), aggregation or except. Products of the type variable can be part of the solution set (i.e. of the fragments $sol_{t,j}$ of the query expression to be generated) and can generate SPARQL statements of the type ?x rdf:type *class* where the *class* information is contained in the information of the variable. Constants are fixed (literal) values from the NLQ, like names or numbers. Products of the type predicate are a set of properties (i.e.,

the properties used in the ontology that may fit the verbal query). Products of type except correspond to negation in the NLQ.

Compound products are either triple, condition or graph products. Triples always consist of a subject which must be an object-valued variable, a predicate, and an object which is also a variable (object- or literal-valued). Note that IRI constants cannot yet exist, since they do not occur in NLQs; and constant values occur only in comparisons in conditions. Triples can be translated directly to SPARQL. Conditions consist of a left product of the type variable, a right product of type variable or constant, and one operator. Products of type graph are basically lists of triples and conditions, but can also contain primitive products that are not yet integrated with the rest of the graph.

For calculating to what extent an agent found a solution component (i.e., a fragment of the query), the partial tasks are valuated as sketched in Table 1.

## 5.3 Nodes and Operations

Each node type implements an operation that corresponds to a single conceptual step. The node types are grouped into the following four categories: reader, generator, relator, and reducer. Node types also have parameters to configure their concrete instances. Parameter settings can be changed by mutations through evolution. Further, nodes can have a *confidence* value which can have the values [evident, derived, necessary]. Each node gets a confidence value assigned when created or mutated and gives it to all products. Some nodes are sensitive to those values and base decisions on them.

The nodes can access the SDD via a SQL database and WordNet via the API [8]. In the following, some of the nodes types are described. For the generated output products, the components are indexed with their provenance; *im* denotes the identifier mappings from the SDD described in Section 5.1.

### 5.3.1 Reader Node Types

Reader nodes receive information from the NLP Core output. Some example for reader nodes are:

**Part of Speech:** This is the most essential node type of all. Its only parameter is, which Part-of-Speech tag is handled by it. A Part-of-Speech node gets the whole set of output POS from NLP Core and if the POS tag of the incoming POS matches the parameter of the node, it generates an nlpdata product with the content {$string_{pos}$, $lemma_{pos}$, $position_{pos}$, $POS_{pos}$, $namedEntity_{pos}$}.

**Synonym:** Such nodes use WordNet to find the terms used in the ontology for a word. Therefore, the nodes maintain a dictionary using each known term *term* of the ontology and querying WordNet for synonyms *syn* of *term*. If an nlpdata{*syn*, ...} is received, the node replaces it by nlpdata{*term*, ...}.

**Proper name:** The idea of this node type is to find a sequence of words in the NLQ which together equal a known identifier in the database, e.g. "Great Britain". For each longest exact match in the input, it combines the input nlpdata products into a single product of type nlpdata.

### 5.3.2 Generator Node Types

Generators turn one product into another type of product using information from the SDD. Some of the more fundamental ones are:

**Class Variable Generator - CVGen:** Such nodes generate variables which range over a class. Therefore they check the string and the lemma of an nlpdata and try to find a matching in the SDD. If it finds a matching class, it generates a var{$name_{nlp}$, $position_{nlp}$, $confidence_{node}$, $ClassName_{SDD}$, false, $POS\ tag_{nlp}$}.

**Identifier Node - IdGen:** While the *CVGen* nodes are responsible for variables ranging over classes, the *IdGen* nodes generate products for identifying a specific instance of a class. Incoming nlpdata is checked for containing a string or lemma which also occurs in the property value of the identifier mapping for a property $name_{im}$. Then it generates *subj*:=var{$name_{nlp}$, $position_{nlp}$, $confidence_{node}$, $domain_{SDD}$, false} describing the class, *pred*:=pred{$name_{im}$, $position_{nlp}$, $confidence_{node}$, $properties_{im}$} for the identifying property, the literal-valued *obj*:=var{$name_{im}$, $position_{nlp}$, $confidence_{node}$, string, true} for the value, a triple{*subj,pred,obj*} containing these three triple parts, a *val*:= const{$name_{im}$, $position_{nlp}$} and condition{*obj*, =, *val*}.

### 5.3.3 Relator Node Types

Relators take two or more products and relate them into a compound product, usually triples or conditions. Such products are *possible* fragments of the final query. The modifier nodes and reducer nodes described below in Sections 5.3.4 and 5.3.5 will remove non-helpful fragments later. E.g. :

**Triple Generator - TriGen:** This node type generates any ontologically possible relationship in form of according triples. For a subject and either a predicate or an object, a filler for the missing position is generated. Either a var{$name_{pred}$, $position_{pred}$, $confidence_{node}$, ranges(*pred*)$_{SDD}$, isLiteral?$_{SDD}$} is created as object, or a pred{$name_{object}$, $position_{object}$, $confidence_{node}$, $properties_{SDD}$} is generated where the properties from the SDD are taken that are defined for the subject's class and where the object's class is in the range.

For literal-valued properties this is often the only way to generate the object since they are not of a class of the ontology and cannot be found by a *CVGen*.

### 5.3.4 Modifier Node Types

Nodes of modifier types perform context-sensitive tasks and have only one input conduit that accepts graph products. An Example for this kind is the

**Reificator:** The goal of nodes of this type is to access the literal values of attributed relations which are usually modeled in RDF through reification. While the terminology of the reified classes is normally not used in NLQ but the direct relation between the entities is used, like "percentage of Russia located in Asia", where "percentage" seems to be a property of countries, and not, as in a reified modeling, of an "EncompassedInfo" resource. The SDD has information about those classes and if encountered, the reificator breaks down the direct relation into the detour over the reified class and generates additionally the predicates for the properties of the reification. The output are *reifvar*:= var{$name_{SDD}$, min($position_A$,$position_B$), $confidence_{node}$, $reified\_class_{SDD}$, false, -}, triple{$Variable_A$, $reifiedPropertyA_{SDD}$, *reifvar*}, triple{*reifvar*, $reifiedPropertyBinv_{SDD}$, $Variable_B$}, and pred{$name_{SDD}$, $position_{view}$, $confidence_{node}$, $property_{SDD}$} that relate $Variable_A$ to a new variable *reifvar* ranging over the reified class etc.

**Table 1: Overview of all Product types.**

| Product | Content | Superclass | success calculation (sketch) |
|---|---|---|---|
| nlpdata | string, lemma, position, named entity tag, POS tag | product | none |
| triplepart | name, position, confidence | product | (abstract class) |
| variable | domain(s), isLiteralValued (t/f), POS tag | triplepart | name + position + domain + all |
| constant | - | triplepart | name |
| predicate | properties | triplepart | name + position + properties + all |
| symbol | - | product | (abstract class) |
| operator | value, position | symbol | value + position |
| aggregation | type, variable | symbol | type + variable + all |
| except | position | symbol | position |
| triple | subject (variable), predicate, object (variable/constant) | product | subject + predicate + object + all |
| condition | left (triplepart), operator, right (triplepart) | product | left + operator + right + all |
| graph | list of products | product | sum of its parts |

#### 5.3.5 *Reducer Node Types*

Reducer nodes reduce the number of products circulating in the agent. Such nodes use the SDD and the context of the products to remove products that are invalid or considered not to be helpful. The following nodes are a selection to demonstrate the general functions of this types.

**Fusion Node - Fus:** Such nodes reduce the domains or properties if more precise information is available. Especially relator nodes often generate two triples describing the same fact, but since either the predicate or the object is inferred, often the properties respectively the domains are too general in the inferred triple part. A fusion node checks a graph product whether it contains triples A and B such that $subject_A$=$subject_B$, $pred_A \subseteq pred_B$ and $objectClass_A \supseteq objectClass_B$, and in this case replaces both triples by triple{$subject_A$, $pred_A$, $object_B$}.

**Conflicting literal solver - CSolv:** Nodes of this type react to graph products with multiple object-valued variables that refer with the same property to a single literal-valued variable. While this is a valid operation in SPARQL, in NLQ this is expressed in a way that would trigger the operator generator, e.g. "where the population is *equal*" or "with the *same* name". In this case, it removes all but one of the conflicting triples, based on the grammatical distance.

### 5.4 Standard Agent

A solid initial basis for the structure of the agents is constructed (automatically) from the information contained in the training set and the application-specific nodes and products. First, for every primitive product type, the set of POS tags and keywords for node parameters to which they can correspond is computed. From this, typical agent substructures for each kind of primitive products, i.e., variables, properties, operators, and excepts are constructed algorithmically.

Next, substructures are generated that depend on how these primitive products are used by relators (for generating triples and conditions). Their output conduits are directly connected to the output node. So far, this is already a very basic agent. At this point already more than half of the possible rewarded energy from the used training set is achieved, but only very simple queries are already sufficiently answered.

For achieving better results, better agents must then evolve from the evolutionary process, where they "learn" to make use of the context-sensitive nodes.

## 6. EVALUATION

The approach has been tested on the Mondial RDF Data set with a set of 51 questions. Only a few of them are simple selections which can be answered in a single SPARQL triple, instead the focus is on more complex and ambiguous questions. The standard agent can answer 45% correctly while the best learned agent was able to give the correct answer to 84% of the questions (examples shown in Table 2). Since the approach is still under development and some key features are still missing, mainly the aggregation functions and the translation from the internal representation into syntactical correct SPARQL, therefore there is no extensive comparison with [6, 4]. The main problems at the moment are the distinction between "and" which mean both sides should have a certain property like query 11 (Table 2 and "and" which mean a union of both sides like query 12. Agents so far only were able to answer correctly one or the other. Another big issue is the lexical gap, as already stated from Steinmetz et al. [10], e.g. the query 13 is answered wrongly because the approach is unable to map inhabitants to the property population and therefore uses a union over all numeric properties of cities. Further logical concepts are not covered at all, like the population density in query 15 (only the properties population and area are existent in the ontology), further the approach is not aware, what it is describing as a whole, therefore in query 16 it does not just list all seas, but tries to find "world" as an instance, does not succeed and completes it to a union over several instances with world in their name like the "World Health Organization" and the "World Trading Union" which are not directly relatable with seas and drifts into complete nonsense. Both are problems mentioned by Saha et al. [6] as well and to the best of our knowledge, these problems have not yet been solved exhaustively for generic cases.

## 7. CONCLUSION

In this paper, we developed an approach that enables agents used in artificial life to work as an functional NLIDB. Therefore we developed a framework to enable those agents to solve complex problems (other than surviving in their environment) which can be broken down into sub-objectives. The agents, which are based on evolutionary programming,

| Nr | NLQ | Correct |
|---|---|---|
| 1 | Give me all rivers with a length shorter than 100 kilometers. | ✓ |
| 2 | List all names except for Deserts. | ✓ |
| 3 | Give me everything located in Asia. | ✓ |
| 4 | Which cities are in Europe? | ✓ |
| 5 | What is the depth of the Sea of Japan? | ✓ |
| 6 | How many percent of India are Sikh? | ✓ |
| 7 | Give me all cities where the population is greater then the population of the capital of their country. | ✓ |
| 8 | Show me all waters with their name | ✓ |
| 9 | Is there a city where the latitude and longitude are equal | ✓ |
| 10 | Is the percentage of Turkish people greater than the percentage of Croat people in Austria | ✓ |
| 11 | Which rivers are located in Poland and Germany? | ✓ |
| 12 | Give me the name of all mountains and islands | ✗ |
| 13 | Give me all cities that have more than 1000000 inhabitants, and are not located at any river that is more than 1000 km long | ✗ |
| 14 | Give me all cities that have a population higher than 1000000, and are not located at any river that is more than 1000 km long | ✓ |
| 15 | How high is the population density in Japan? | ✗ |
| 16 | How many seas are there in the world? | ✗ |

**Table 2: Example queries from the test set**

had to be extended and transferred from a linear to a multi-dimensional evaluation system to cope with the complexity of NLQ processing. For this purpose an evaluation technique, which not only takes into account the agents with the highest score, but also those who have specialized in a new direction and thus extend the functionality of the whole approach. The agents have been equipped with specialized operations for their architecture, but also with many common ontological or graph pattern based operations and are able to link them in a meaningful way to transform them into an NLIDB. The intermediate results, although not yet final, are comparable with existing approaches. Since some featrues are still missing and the evaluation is not executed in SPARQL but in the internal query language for the moment, this is not precisely comparable. But it gives reasons for the assumption that this approach might be comparable to other state of the art approaches and might also provide additional flexibility in some cases.

# 8. REFERENCES


[1] S. Auer, C. Bizer, G. Kobilarov et al. Dbpedia: A nucleus for a web of open data. In *ISWC*, Springer LNCS 4825, pages 722–735. 2007.

[2] T. Hoverd and S. Stepney. Energy as a driver of diversity in open-ended evolution. In *ECAL 2011*, pp. 356-363, ACM. 2011.

[3] C. D. Manning, M. Surdeanu, J. Bauer et al. The Stanford CoreNLP natural language processing toolkit. In *ACL*, pages 55–60, 2014.

[4] A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *Intelligent User Interfaces*, pp. 149–157. ACM, 2003.

[5] L. Runge, S. Schrage, and W. May. Systematical representation of RDF-to-relational mappings for ontology-based data access. Technical report, available at `https://www.dbis.informatik.uni-goettingen.de/Publics/17/odbase17.html`, 2017.

[6] D. Saha, A. Floratou, K. Sankaranarayanan et al. Athena: An ontology-driven system for natural language querying over relational data stores. *VLDB*, 9:1209–1220, 2016.

[7] The Mondial database. `http://dbis.informatik.uni-goettingen.de/Mondial`.

[8] C. Fellbaum (1998, ed.) WordNet: An Electronic Lexical Database. MIT Press.

[9] S. Hu, L. Zou, X. Zhang. A State-transition Framework to Answer Complex Questions over Knowledge Base. *EMNLP*, pp. 2098-2108, 2018.

[10] N. Steinmetz, A. Arning, K.U. Sattler From Natural Language Questions to SPARQL Queries: A Pattern-based Approach. *BTW* pp. 289-308. LNI, 2019

[11] G. Turk. Sticky feet: Evolution in a multi-creature physical simulation. *InALife XII*, pages 496-503. MITPress, 2010.

[12] P. A. Vikhar Evolutionary algorithms: A critical review and its future prospects. *ICGTSPICC*. 2016, pp. 261-265.

[13] Y.-L. Li, Y.-R. Zhou, Z.-H. Zhan, J. Zhang, "A primary Theoretical Study on Decomposition Based Multiobjective Evolutionary Algorithm", IEEE, Volume: 20, Issue: 4, pp. 563-576, 2015