ValueNet4SPARQL: Constructing, Enriching and Querying Knowledge Graphs in Natural Language

Catherine Kosten a,b,*, Ursin Brunner a, Diego Calvanese b,c, Philippe Cudre-Mauroux d, Davide Lanti b, Alessandro Mosca b and Kurt Stockinger a

a Zurich University of Applied Sciences, Winterthur, Switzerland
b Free University of Bozen-Bolzano, Bolzano, Italy
c Umeå University, Umeå, Sweden
d University of Fribourg, Fribourg, Switzerland

Abstract. As Knowledge Graphs (KGs) gain traction in both industry and the public sector, more and more legacy databases are accessed through a KG-based layer. Querying such layers requires the mastery of intricate declarative languages such as SPARQL, prompting the need for simpler interfaces, e.g., in natural language (NL). However, translating NL questions into SPARQL and executing the resulting queries on top of a KG-based access layer is impractical for two reasons: (i) automatically generating correct SPARQL queries from NL is difficult as training data is typically scarce and (ii) executing the resulting queries through a simplistic KG layer automatically derived from an underlying relational schema yields poor results.

To solve both issues, we introduce ValueNet4SPARQL, an end-to-end NL-to-SPARQL system capable of generating high-quality SPARQL queries from NL questions using a transformer-based neural network architecture. ValueNet4SPARQL can re-use neural models that were trained on SQL databases and therefore does not require any additional NL/SPARQL-pairs as training data. In addition, our system is able to reconstruct rich schema information in the KG from its relational counterpart using a workload-based analysis, and to faithfully translate complex operations (such as joins or aggregates) from NL to SPARQL. We apply our approach for reconstructing schema information in the KG on the well-known data set Spider and show that it considerably improves the accuracy of the NL-to-SPARQL results—by up to 36% (for a total accuracy of 94%) —compared to a standard baseline. Finally, we also evaluate ValueNet4SPARQL on the well known LC-QuAD 1.0 data set and achieve an F1-score of 85%, which outperforms the state-of-the-art system by 17%.

Keywords: Knowledge graph construction and enrichment, ontology-based data access, natural language questions

1. Introduction

More data than ever before is being collected and stored in databases (DBs) to put to use for individuals and businesses alike. Until recently most data was stored in relational databases, however, Knowledge Graphs (KGs) [20] have started to gain significant industrial traction across several domains. While having ever more data at our fingertips is pushing many industries forward, much of the data that has been collected is not accessible to the average individual because they do not have the necessary skill set to gather and analyze this data. Natural Language (NL) interfaces to DBs are advantageous because they free users from the need to know the query language and (to some degree) the precise organization of the data itself. Consequently the interest in building systems that translate NL queries to the native language of the underlying DB systems has grown significantly [1].

Creating more robust NL interfaces to databases is a problem that has plagued the database research community for decades [2, 9, 28]. However, in the notable context of relational DBs, recent research shows
significant improvements in NL interfaces through the implementation of deep learning techniques [24], which in turn have motivated the creation and use of large-scale, cross-domain training resources [25]. Deep learning models that have been trained on these data sets can be reused and generalize well to previously unseen data sets. Indeed, NL-to-SQL systems already perform very well, as evidenced by the most recent entries on the well-known Spider Text-to-SQL challenge leaderboard\(^1\). For instance, in the Execution with values category, the best approach currently performs with 78.5% accuracy.

One of the most popular query languages for KGs is the W3C recommendation SPARQL query language [19]. Despite the high-level conceptualization typical of KGs, which allows domain experts to query the data sources transparently, SPARQL still poses significant challenges for the use by people who are not trained computer scientists. Hence, it is also important to provide NL interfaces for KG-based systems accessible through SPARQL. This is, however, only addressed in a limited number of works (e.g., [37]). A major hindrance to the adoption of deep learning based systems for KGs is the lack of resources to train them. Because developing new high-quality and labelled data sets for deep learning is extremely time consuming and costly, we propose leveraging existing resources (and systems) developed for NL-to-SQL systems.

ValueNet [8] is an NL-to-SQL system, that generates SQL queries from NL questions using a transformer-based neural network architecture. The system on which ValueNet is based, IRNet [17], is comprised of a grammar-based intermediate language, SemQL, to which NL queries are mapped, and a deterministic translation of SemQL to the final SQL statement. SemQL offers general data manipulation operations that are not tied in any way to SQL. In this sense, it is query-language agnostic and can be used to generate queries in a different target language, such as SPARQL. For this reason, ValueNet appears to be an ideal system to test the effectiveness of using NL-to-SQL resources/systems to produce NL interfaces for KGs, in this case more specifically, NL-to-SPARQL.

In this paper, we build on the above two main ideas (re-using NL-to-SQL systems and training data and applying them to SPARQL and improving existing training data automatically), and exploit ValueNet by training it using the Spider data set [38]\(^2\), which is considered the de-facto standard for training NL-to-SQL systems, and adapt it to SPARQL as target query language. Such adaption is non-trivial since Spider, despite its popularity, presents several modeling issues that make it non-conformant to best practices in database design (and even to the SQL standard, as far as queries are concerned).

Most notably, 10% of the Spider databases have at least one table that is missing a foreign key constraint. As an example, the Spider database flight_2, shown in Figure 1, is missing a primary/foreign key between the tables flights and airlines. We leverage the existing Spider relational databases by converting them to knowledge graphs with Direct Mapping (described in Section 2.1). To remedy the issue of missing foreign key constraints, we analyzed all queries from the data set containing joins, since these provide hints as to where relationships between tables should be expected. By exploiting these queries, we were able to detect missing foreign key relationships and to add this information to the converted KG’s, thus creating enriched knowledge graphs.

More specifically, in this paper we provide the following contributions:

- We present a general methodology on how to adapt existing training resources for NL-to-SQL systems to the NL-to-SPARQL setting (see Section 3). The methodology automatically detects and corrects data modelling issues, by analyzing both query workloads and data (see Section 4).
- We adapt the target query generation techniques of the ValueNet system to SPARQL, and imple-

---

\(^1\)Spider Text-to-SQL Challenge: https://yale-lily.github.io/spider

\(^2\)https://github.com/taoyds/spider
ment them in the novel ValueNet4SPARQL system (see Section 5).

- We apply our methodology to the Spider data set, thus obtaining a new benchmarking data set for NL-to-SPARQL systems (see Section 6).

- We perform an experimental evaluation to validate our approach, by executing the automatically generated SPARQL queries on two different versions of the knowledge graphs (baseline and enriched) and achieve execution accuracies of 58% and 94%, respectively (see Section 6.3).

- To demonstrate the generalizability of our system, we evaluate ValueNet4SPARQL on the well known LC-QuAD 1.0 data set and achieve an F1-score of 85% outperforming the state-of-the-art system by 17% (see Section 6.4).

- The significant improvement in performance of ValueNet4SPARQL on the LC-QuAD 1.0 data set indicates that our system is not only query-language agnostic (we can use data from both SQL and SPARQL data sets) but also that it is generalizable to a variety of knowledge graph structures and formats (the graph structure of the knowledge graph behind lcquad and the converted spider data sets differ significantly).

As a by-product of our research efforts, we obtain improved data models and an overall cleaner version of the Spider dev set in the PostgreSQL format, and make it available to the community.3

The paper is structured as follows. Section 2 gives a high level overview on the end-to-end process for generating, enriching and querying knowledge graphs. The details on knowledge graph construction, knowledge graph enrichment and querying knowledge graphs in natural language are given in Sections 3, 4 and 5, respectively. In Section 6, we perform a detailed experimental evaluation of our system based on two different data sets. Related work is discussed in Section 7 followed by conclusions in Section 8.

2. End to End Process of Generating, Enriching and Querying Knowledge Graphs

In this section we give a high level overview of the methods used to construct and enrich knowledge graphs as well as a brief description of the NL-to-SPARQL system ValueNet4SPARQL. An overview of the end-to-end-process is shown in Figure 2. The green boxes and arrows show the major contributions of this paper. The blue boxes and arrows indicate existing solutions. The main steps are (1) Knowledge Graph Generation, (2) Knowledge Graph Enrichment and (3) Translating a natural language question to SPARQL. These steps will be discussed in detail below.

2.1. Knowledge Graph Generation

To build a KG based on an existing DB, we apply the Direct Mapping [3] approach. Direct Mapping is a W3C recommendation stipulating how to transform a DB into a KG, while preserving the information and the vocabulary used in the DB. The fact that the vocabulary is preserved allows us to have a direct correspondence between the SQL DB and the produced SPARQL KG. This puts us in the advantageous setting where we can directly apply well-established techniques developed for the SQL setting to the SPARQL case. A detailed description of our KG construction process is given in Section 3.3.

2.2. Knowledge Graph Enrichment

A major difference between a DB and a KG created through Direct Mapping is the fact that the latter does not include a schema. Hence, all the information present in the DB schema is essentially lost during the transformation to the KG. This behavior of

3https://github.com/ckosten/Spider_psql_kg_resources
Direct Mapping is justified since the recommendation is specifically designed to be agnostic of any ontology language that might be adopted for the constructed KG.

Another limitation of Direct Mapping is the fact that, when the DB schema constraints are under-specified, e.g., a primary/foreign key relationship is missing, the generated KG lacks crucial information, such as the relations between objects belonging to different classes. To overcome this limitation, we devised an approach based on query workloads, which allows us to reconstruct the missing schema information and, in turn, deliver a more faithful KG representation of the information stored in the DB. In summary, we analyze the SQL query workloads against the original DB and use this knowledge to enrich the KG. A detailed description of our contributions of this process is given in Section 4.

2.3. Translating a Natural Language Question to SPARQL

The next step, after generating and enriching the KG, is to translate a natural language question to SPARQL and thus to query the knowledge graph. Hence, we introduce ValueNet4SPARQL, an end-to-end natural language to SPARQL system that leverages components from the neural network-based NL-to-SQL system ValueNet. The advantage of ValueNet4SPARQL is that it can re-use neural models that are trained on SQL databases and therefore does not require any additional NL/SPARQL-pairs as training data.

ValueNet4SPARQL takes an NL question and a KG as input, and produces a SPARQL-statement that encodes the NL question. In the pre-processing step, matches are found between tokens in the NL question and classes and properties of the KG Schema. The output of the pre-processing step is fed to a transformer-based neural network architecture, which translates the NL question into the intermediate language SemQL. Finally, the SemQL-based query representation is translated into SPARQL.

The differences in the system architecture of ValueNet and ValueNet4SPARQL are shown in Figure 3. The main contributions of the paper are marked in red.

3. Constructing Knowledge Graphs

3.1. RDF – A Schema-Less Language

The Resource Description Framework (RDF) [10] format is the W3C standard for Knowledge Graphs (KGs) [20]. An RDF graph is a set of triples of the form subject-predicate-object, where each triple expresses a simple sentence (e.g., “Ribbon is a cat.”) or quantified sentences (e.g., “Every tractor is a machine.”). SPARQL [19] is the W3C standard for querying RDF graphs.

As opposed to traditional relational databases, RDF (similarly to other KG languages) is a schema-less language. This enables the flexibility necessary in all con-
To model structured information, users rely on extended RDF vocabularies, such as RDFS [7] or one of the OWL2 [5] profiles. Given that the OWL2 profile specifically designed for data-intensive applications is OWL2 QL [27], we focus our attention on this specific profile.

3.2. Classes, Data Properties, and Object Properties

In OWL 2 QL, information is structured according to three pairwise-disjoint sets of classes, data properties, and object properties. Similarly to the object-oriented paradigm, classes specify sets of objects (e.g., the class Airport containing specific instances of airports). Data properties are used to connect objects to RDF literals (e.g., the country of the airport ASY is the string value “United States”). Object properties relate pairs of objects (e.g., the SourceAirport relationship between flights and airports). Figure 4 shows a portion of a KG representing the data in the flight_2 database, with the data property edges labelled in green and object property edges labelled in blue.

OWL 2 QL is the language of choice for the Virtual Knowledge Graphs (VKGs) approach [35, 36], depicted in Figure 5. In such an approach, a legacy (relational) data source is exposed as a virtual RDF graph. This is done by relating classes and properties in an OWL2 QL ontology to SQL queries over the data sources through a declarative mapping.

The KG is virtual in the sense that RDF triples that are executed against the (possibly federated) data source directly. Systems employing such an approach, e.g., the popular open-source system Ontop5, are known as VKG systems. Another approach is instead to physically store the RDF triples using a triple store like Stardog6, in which case we say that the KG is materialized. The advantage of the virtual approach is the guarantee that the data to be accessed is always up-to-date, whereas the materialized approach can be advantageous in contexts where the freshness of the data is not important but some performance guarantees on query answering are required.

3.3. Mapping of Relational Schema to RDF Knowledge Graph

The W3C provides two (not fully compatible) standards for mapping relational sources to RDF graphs. The first one, Direct Mapping [3] is a W3C recommendation which operates in a fully automatic way starting from a database schema: each table becomes a class in the ontology, each attribute becomes a (data) property relating objects in the class to RDF literals (e.g., strings, integers, etc.), and each foreign key re-
4. Knowledge Graph Enrichment

In this section we discuss a number of new techniques to overcome the limitations of Direct Mapping pointed out in the previous section. The techniques were implemented in a tool called MPBoot, which exploits the flexibility provided by R2RML. Such tool extends the bootstrapper built in the Ontop VKG system, where a bootstrapper is a system which automatically generates an initial (R2RML) mapping and ontology starting from a DB. Apart from a few specific corner-cases (which we are going to analyze in the next subsections), the KG obtained through the Ontop bootstrapper follows the Direct Mapping specification for most part.

4.1. Encoding Foreign Keys through OWL2QL

In Direct Mapping, each foreign key relating two tables is translated into an object property relating objects between the classes corresponding to the tables participating in the relationship. We actually have already encountered an example of Direct Mapping, specifically when discussing the KG version of the flights_2 DB displayed in Figure 4.

One drawback of Direct Mapping is that it requires the source schema to be sufficiently structured (i.e., primary and foreign key constraints must be explicitly declared).

R2RML [11] is a W3C recommendation for mappings that overcomes this limitation. Through R2RML, users can specify mappings by hand. This allows for greater flexibility because the data source can be mapped to arbitrary ontologies, regardless of the vocabulary and the structure of the data source. However, in the virtual approach, this comes at the price of sacrificing performance since the knowledge graph is not materialized but derived through mappings during query processing.

4.2. Dealing with Missing Constraints with Query-driven KG Enrichment

Consider again the flight_2 schema displayed in Figure 1. Note that table airlines is not related to any other table of the DB. Observe, though, that the presence of an attribute Airline in table flights provides a hint on the fact that the schema might be under-specified, that is, that a foreign key is actually present between flights and airlines but it has not been explicitly declared in the DB schema. We propose a method, based on query workloads, to automatically detect such defective schemas and fix them in their transposition to KGs.

One effect of this is that it is not possible to infer the direction of the property between the two classes starting only from the ontology axioms. Recall that the direction of an object property is crucial in understanding a KG, as we will further explain in Section 5.1.1. Observe that this is purely a limitation of Direct Mapping, which fails at preserving the direction specified in the database schema by the "referring" and "referred" sides of a foreign key. By exploiting the expressiveness of OWL 2 QL, and similarly to the approach described in [21], MPBoot enriches the KG with suitable domain and range axioms, preserving thus the direction specified in the database schema.

Consider again the flight_2 KG displayed in Figure 1. Note that table airlines is not related to any other table of the DB. Observe, though, that the presence of an attribute Airline in table flights provides a hint on the fact that the schema might be under-specified, that is, that a foreign key is actually present between flights and airlines but it has not been explicitly declared in the DB schema. We propose a method, based on query workloads, to automatically detect such defective schemas and fix them in their transposition to KGs.
left). Observe that the only existing connection between flight A1N28 and airline A1 is the fact that the airline number of A1N28 coincides with the uid value of A1. In principle this might be purely coincidental, and that it is not enough to establish a connection between objects A1 and A1N28 (the red, dotted arrow in Figure 6).

Since we cannot rely on Direct Mapping, in MPBoot we exploit the flexibility of R2RML to remedy this situation. MPBoot uses a SQL query workload as a guide to produce a better KG despite missing schema constraints. We show how MPBoot operates in the following example.

**Example 1.** Consider the following user query over the flight_2 DB:

```
SELECT A.abbreviation
FROM airlines A JOIN flights F
ON A.uid = F.airline
```

The join condition between flights and airlines hints that there exists an undeclared foreign key between the two tables. Based on this, MPBoot checks:

- whether either attribute is declared as UNIQUE or PRIMARY KEY in the DB schema. In our example, uid satisfies this condition, but not airline,
- whether the containment towards the UNIQUE attribute holds in the DB data. In our example, all values for airline are actually also values in uid and vice-versa.

Since all conditions for a foreign key from airline to uid are satisfied, MPBoot creates an object property (and relative domain and range axioms) connecting flights and airlines. Hence, the hasAirline edge between A1N28 and A1 will correctly appear in the final KG.

We now provide the details of the algorithm, which is based on the notion of join-pair.

**Definition 1.** A join-pair is an expression of the form $A \rightarrow B$, where:
- $A$ and $B$ are duplicate-free sequences of attributes;
- $A$ and $B$ have the same length;
- the attributes in $A$ (resp., $B$) are all over the same table.

```
Input : A set $Q$ of SQL queries
Output: A set $J$ of join-pairs
foreach Query $q \in Q$ do
    eqs = extractEqualities($q$);
    groups = groupAndSortEqualities(eqs);
    foreach Group $g \in groups$ do
        JoinPair $jp$ = makeJoinPair($g$);
        $J = J \cup \{jp\}$
    end
end
return $J$:
```

Algorithm 1: Extraction of the join-pairs enabling query-driven knowledge graph enrichment.

Intuitively, a join-pair $A \rightarrow B$ indicates a candidate foreign key where the attributes in $A$ are the referencing attributes, and the attributes in $B$ are the referenced attributes.

Algorithm 1 shows the extraction of join-pairs starting from a set of workload queries. Given a query, the function `extractEqualities` extracts all equalities between pairs of attributes occurring in that query, and the function `groupAndSortEqualities` groups together all equalities involving the same pair of tables, and sorts the equalities arguments according to an arbitrary table order. Finally, given a group $\{s_1 = t_1, \ldots, s_n = t_n\}$ of equalities over two tables $S$ and $T$, the function `makeJoinPair` creates a join-pair $A \rightarrow B$, with $A = (s_1, \ldots, s_n)$ and $B = (t_1, \ldots, t_n)$.

Over the extracted join-pairs, MPBoot then checks the data and schema conditions as in Example 1, and creates additional object properties if such conditions are met.

### 4.3. Dealing with the Absence of Primary Keys

As we have seen, Ontop can be used to bootstrap an initial set of R2RML mappings which, together with

```
<table>
<thead>
<tr>
<th>ranking_date</th>
<th>ranking</th>
<th>player_id</th>
<th>ranking_points</th>
<th>tours</th>
</tr>
</thead>
<tbody>
<tr>
<td>20170814</td>
<td>509</td>
<td>NULL</td>
<td>55</td>
<td>15</td>
</tr>
<tr>
<td>20000101</td>
<td>1</td>
<td>200748</td>
<td>4378</td>
<td>13</td>
</tr>
<tr>
<td>20000101</td>
<td>4</td>
<td>200033</td>
<td>3021</td>
<td>15</td>
</tr>
</tbody>
</table>
```

Fig. 7. A sample from the rankings table of the database wta_1.

Intuitively, a join-pair $A \rightarrow B$ indicates a candidate foreign key where the attributes in $A$ are the referencing attributes, and the attributes in $B$ are the referenced attributes.
Fig. 8. RDF knowledge graph generated by the Ontop Direct Mapping starting from Table 7. Note that the first tuple shown in Table 7 is missing.

However, the solution also brings disadvantages. ODM is not able to deal with the cases where one of the tables is without a primary key and contains NULL values. This is due to the very nature of the bootstrapper being an R2RML-based and fully-automated process. In the wta_1 DB from the Spider data set, the table rankings (a portion of which is displayed in Figure 7) is one such table.

Figure 8 shows the triples generated by the R2RML mapping produced by Ontop, and the data displayed in Figure 7. As we can see, no object for capturing the first tuple in the rankings table is produced. Intuitively, the R2RML mapping produced by Ontop prescribes that a resource identifier (i.e., the gray nodes in Figure 8) must be built for each tuple in the table. For the case of a table without a primary key, ODM creates resource identifiers as blank nodes built out of all the values in a tuple. However, the R2RML specification forbids the creation of a resource identifier out of NULL values. For such a reason, ODM does not produce a resource identifier corresponding to the first tuple of table rankings. As a result, the transformation from the relational format to the graph format is lossy, as the information contained in the first tuple is not correctly represented in the final KG.

In MPBoot, we solve this problem by adopting a semi-automatic bootstrapping strategy: the user specifies default values for NULL, according to the attribute datatype. MPBoot will then use the values specified by the user every time a NULL is required to construct a resource identifier.

Figure 9 shows the knowledge graph produced by applying our semi-automatic bootstrapping strategy, where the default value for NULL has been set to the string “NIL”. Observe that this knowledge graph now correctly represents the information that was present in the original rankings table.

5. Querying Knowledge Graphs in Natural Language

In this section we detail the steps used in ValueNet4SPARQL to translate a natural language question to SPARQL. This translation occurs in 2 steps. Step 1: Translate an NL question to the intermediate query representation SemQL using neural networks. Step 2: Translate SemQL to SPARQL using a deterministic approach. Our main focus and the novelty of our approach lie in Step 2.

Since in our approach we use the same intermediate query representation SemQL both for SQL and SPARQL queries, in order to introduce the translation of Step 2, we first need to discuss certain relevant differences between SQL and SPARQL.

5.1. Query Processing: SQL vs. SPARQL

In this section we describe the differences in query processing between SQL and SPARQL.
only focus on those aspects that are relevant for translating NL to SPARQL, such as handling of joins, aggregation queries and set operation queries. Thus, do not cover the full specification of these query languages.

5.1.1. Syntactic vs. Conceptual Joins

An important difference between SPARQL and SQL is the fact that the latter is purely syntactic: relations are tuples of values, and there is no high-level conceptual notion such as object or class. Consider, for instance, the following SQL query over the flight_2 database from the running example:

```sql
SELECT City
FROM airports AS t1
JOIN flights AS t2
ON t1.AirportCode = t2.SourceAirport
```

A conceptual interpretation of the SQL query is “Retrieve the cities of all airports that are also source airports”. However, the literal interpretation of such a query, without adding a semantic layer to it, would actually be similar to “Retrieve all the values of the City attribute in table airports such that the value of AirportCode is syntactically equal to some value of SourceAirport in table flights.”

The conceptual interpretation we gave above comes from the semantics we usually attach to a relational database, which is based on our knowledge of the domain and on the schema of the relational database. In fact, such a schema is usually derived starting from a well-thought high-level conceptualization based on ER-diagrams.

Consider the database schema of our running example from Figure 1. Given such a schema, we can say that the syntactic join between airports and flights represents the actual conceptual relationship between airports and flights. The reason is that AirportCode identifies instances of airports and the foreign key between AirportCode and SourceAirport is the result of translating such an ER-diagram into the (non-conceptual) relational model.

A key difference between SPARQL and SQL is that SPARQL is able to naturally formulate such conceptual joins:

```sparql
SELECT ?t1_city WHERE {
  ?t1 a dbo:airports . # can be omitted
  ?t2 a dbo:flights . # can be omitted
  ?t2 dbo:flights#ref-sourceairport ?ac .
  ?t1 dbo:airsports#airportcode ?ac .
  ?t1 dbo:airsports#city ?t1_city.
}
```

In the SPARQL query above, the conceptual relationship between flights and airports is captured explicitly through the dbo:flights#ref-sourceairport object property. Observe that, since RDF is a directed graph, it is important to know what is the direction of the relationship between flights and airports.

In other words, it becomes important to know that the domain of dbo:flights#ref-sourceairport is a flight, and that the range of the same property is an airport. Fortunately, OWL 2 QL is rich enough to allow for the definition of such crucial domain and range information, and we will rely on it throughout the rest of this paper.

Note that value-based (syntactic) joins can be expressed in SPARQL as well. With reference to our example, this means that flights and airports can be combined even without an explicit object property linking the objects of the two classes. Such combinations are supported by means of SPARQL ON joins, which replicate the SQL (syntactic) join, that is, by enforcing the equality between the values of the dbo:flights#sourceairport and dbo:flights#sourceairport properties:

```sparql
SELECT ?t1_city WHERE {
  ?t1 a dbo:airports .
  ?t2 a dbo:flights .
  ?t1 dbo:airsports#airportcode ?ac .
  ?t2 dbo:flights#sourceairport ?ac .
  ?t1 dbo:airsports#city ?t1_city.
}
```

Observe that the query above corresponds to our example SQL query only if the identification of objects by means of attributes is carried over to the RDF triples.

5.1.2. Aggregation Queries

The following example shows that SQLite (the DB engine used in the original Spider data set) allows queries with aggregated and non-aggregated variables in the query projection. For instance, the attribute `t1.City` is part of the projection but not in the `GROUP BY` clause:

```sql
SELECT t1.City, count(*)
FROM airports AS t1
JOIN flights as t2
ON t1.AirportCode = t2.SourceAirport
```

#Direct Mapping guarantees this property by enforcing the use of percent-encoding when building resource identifiers. For details, please refer to https://www.w3.org/TR/rdb-direct-mapping/.
GROUP BY t2.SourceAirport

Note that the query above is not compliant with the SQL standard. Likewise, SPARQL cannot execute this type of query as is and returns a MALFORMED QUERY error. According to the SPARQL 1.1 Recommendation, queries that have aggregates in their projection may only include non-aggregated variables when these are included in a GROUP BY statement\(^9\).

Therefore, in order to execute this type of query in SPARQL, the query needs to be rewritten as follows:

```
SELECT ?t1_city (count(*) as ?aggregation_all)
WHERE {
  ?t1 a dbo:airports .
  ?t2 a dbo:flights .
  ?t2 dbo:flights#ref-sourceairport ?t1 .
  ?t1 dbo:airports#city ?t1_city .
  ?t2 dbo:flights#sourceairport ?t2_sourceairport .
}
GROUP BY ?t1_city ?t2_sourceairport
```

The query above is now executable because the variables in the projection are now included in the GROUP BY statement.

5.1.3. Set Operation Queries

Certain SQL keywords do not exist in SPARQL, such as INTERSECT, EXCEPT, and BETWEEN. In order to achieve these kinds of set operations in SPARQL, alternative formulations are required. Consider the following SQL example taken from the Spider data set:

```
SELECT City FROM airports WHERE AirportCode = 'MMI'
INTERSECT SELECT City FROM airports WHERE AirportCode = 'AHN'
```

The corresponding SPARQL query is as follows:

```
SELECT ?t1_city WHERE {
  ?t1 a dbo:airports .
  ?t1 dbo:airports#city ?t1_city .
  ?t1 dbo:airports#airportcode ?t1_airportcode .
  FILTER(?t1_airportcode = 'MMI') .
  FILTER(?t1_city IN (?t2_city)) .
}
```

The SQL-INTERSECT construct can be replicated in SPARQL by adding a FILTER-IN-clause for the variable result that mirrors the projected columns in the SELECT statement in the SQL query. This clause checks if the results from the variable ?t1_city are also in the results from the variable ?t2_city.

5.1.4. Query-Driven Datatype Conversion

In SQLite, it is possible to perform a `sum` or `avg` aggregation over a column that has numeric values stored as a string. There are many columns across the databases in the Spider development set that have numeric data stored as text datatypes such as `varchar`. This level of datatype flexibility is not supported in PostgreSQL or SPARQL. Such a query returns an error in PostgreSQL and an empty set in SPARQL. In order to correct these incorrectly stored data, we performed a query-driven datatype conversion. For each query that contained an aggregation, we evaluated the declared datatype for the aggregated column and converted it accordingly.

5.2. Translating Natural Language to SPARQL

In Section 5.1 we introduced query processing differences between SQL and SPARQL at a conceptual level. We will now explain how we designed and implemented the translation from natural language to SPARQL by considering the differences between the two query languages.

Note that since we leverage ValueNet for our implementation, we use ValueNet’s neural network architecture for translating from natural language to the intermediate query language SemQL (see Figure 3). Therefore, with ValueNet4SPARQL we focus on the translation from SemQL to SPARQL.

Table 1 shows SemQL 2.0, the intermediate language for translating a natural language question to SQL with ValueNet [8]. The highlighted parts of the grammar indicate keywords from SQL that do not have an equivalent in SPARQL. These were translated with substitution keywords from SPARQL or a combination of SPARQL operations which reflect the functionality

\(^9\)https://www.w3.org/TR/sparql11-query/#groupby
of these SQL operations. No additions were required to the SemQL 2.0 grammar for the SPARQL translation.

In order to translate SemQL to SPARQL, the following steps are required:

- Implementing join queries: In order to translate SQL joins into equivalent SPARQL statements, ValueNet4SPARQL takes the domain and range axioms in the enriched KG into account.
- Implementing aggregation queries: Queries with aggregated and non-aggregated variables in the projection are handled by inferring adequate GROUP BY clauses.
- Implementing set operation queries: Queries with set operations that are not available in SPARQL were adapted with either a replacement keyword, such as MINUS, or a complementary set of operations that yield the same result as the SQL set operation keyword.

In the remainder of this section we detail the first two of these steps. The third step is not explained in detail since the implementation is straightforward, similar to the example given in Section 5.1.3.

5.2.1. Implementing Join Queries

Due to the fact that joins in SQL are commutative, the original version of ValueNet did not take into account the order in which the tables to be joined were declared in the SQL query. ValueNet inferred joins, in the post-processing step, shown in Figure 3, by traversing the database schema (which can be described as an graph), to find the shortest path between the predicted tables in the SemQL sketch [17]. The SemQL sketch is the representation of the query using the grammar described in Table 1.

In ValueNet4SPARQL, however, we could not apply the exact same strategy, because the original schema information is not directly available in the KG. In Section 5.1.1 we mention that, if Direct Mapping is used, a SQL (value-based) join can be encoded through a combination of SPARQL joins. However, this encoding requires knowledge about what attributes were involved in a primary key/foreign key relationship. This information is not captured by the KG “schemas” of our setting, even for enriched KGs, since the OWL 2 QL language does not allow for expressing key constraints. In fact, adding such constraints to the language would immediately make it lose its computational complexity guarantees. Hence, our join-inference algorithm can only rely on conceptual joins, which also happen to be the natural way in which people express joins in SPARQL.

Recall that conceptual joins are realized through object properties. Since RDF KGs are directed labeled graphs [20], such properties have a direction. Hence, for the join-inference step to generate the right conceptual joins, this aspect needs to be taken into account.

We show how these problems were addressed by means of our running example about flights from Figure 1. Consider the following SQL query:

```
SELECT t1.City
FROM airports AS t1
JOIN flights AS t2
ON t1.AirportCode = t2.SourceAirport
ORDER BY t2.SourceAirport DESC LIMIT 1
```

Such a query returns the same result set, regardless of whether the tables are joined as t1.AirportCode = t2.SourceAirport or t2.SourceAirport = t1.AirportCode.

Consider again the 1:N relationship between Airports and Flights previously shown in Figure 1. In this case, Flights.SourceAirport is the foreign key that references the primary key AirportCode in table Airports.

If Direct Mapping is used, the direction of such a foreign key affects the direction of the relative object property encoding it. In our setting of enriched KGs, the direction of object properties can be derived from their relative domain and range axioms. Therefore, in
the join-inference step in ValueNet4SPARQL, we not only traverse the graph to find the shortest path between classes, but we also use attributes (domains and ranges) from the enriched KG to infer the direction of the properties between the classes.

The SPARQL query corresponding to the aforementioned SQL query is as follows. Note that the direction of the vertex between the nodes airports and flights is expressed via the predicate dbo:flights#ref-sourceairport in line 4.

```
SELECT ?t1_city WHERE {
  ?t1 a dbo:airports .
  ?t2 a dbo:flights .
  ?t2 dbo:flights#ref-sourceairport ?t1 .
  ?t1 dbo:airports#city ?t1_city .
  ?t2 dbo:flights#sourceairport
  ?t2_sourceairport .
}
ORDER BY DESC (?t2_sourceairport) LIMIT 1
```

The handling of join queries is described in Algorithm 2.

```
Input : A KG G consisting of classes C (nodes) connected with properties with domains and ranges (directed labelled edges)
Input : A set of predicted classes C’ of SemQL
Output: Shortest path P in G between all classes in C’
g ← extractOntology(G)
foreach c_i and c_{i+1} ∈ C’ do
  if c_i and c_{i+1} are in g then
    p ← Dijkstra(c_i, c_{i+1});
    P = P ∪ p
  if check(P, C’) then
    return P;
end
return no_path_found;
```

Algorithm 2: Inferring the join path in a directed graph between two or more classes predicted in a SemQL sketch.

Algorithm 2 shows the method by which joins are determined from the classes predicted in the SemQL sketch. First, the function extractOntology takes the KG G as input and builds a directed labelled graph g in the following way: nodes in g are the classes in G, and a directed edge in g connects the two nodes c_1 and c_2 with a property label l if G contains the axioms stating that l is an object property of domain c_1 and range c_2. Given a set of classes C’, each class in the set is compared against the classes in g. If class c_i exists in g, the Dijkstra algorithm will find the shortest path between c and g. The function check determines whether or not P is a path between all of the classes.

5.2.2. Implementing Aggregation Queries

As described in Section 5.1.2, SPARQL cannot execute queries that have both aggregated and non-aggregated variables in the projection, unless the non-aggregated variables are present in the GROUP BY clause. Algorithm 3 describes the new method for inferring GROUP BY clauses. This method is applied to all queries that have an aggregation in their projection. If there is more than one variable in the projection, we iterate through all of the projected variables and check if the variable is aggregated or not. Each variable in the projection list that is not aggregated is then added to the GROUP BY clause.

```
Input : A set S of variables and aggregates
Output: A set GroupBy for GROUP BY clause
if S contains aggregates AND |S| > 1 then
  foreach s ∈ S do
    if s is a non-aggregated variable then
      GroupBy = GroupBy ∪ \{s\};
  end
end
return GroupBy;
```

Algorithm 3: Determining variables for the GROUP BY-clause for handling aggregation queries.

6. Experiments and Results

In this section we introduce the experimental evaluation of ValueNet4SPARQL and our KG enrichment process. First, we describe the specifics of the data set and the evaluation metrics used. Then, we show a detailed analysis of the efficacy of the KG enrichment via the execution accuracy of SPARQL queries translated with ValueNet4SPARQL. For reproducibility of
therefore, in this evaluation we use result set matching.

One weakness typically associated with this method is that two different SQL queries could produce the same results by chance. Since we are not evaluating NL-to-SPARQL but rather the accuracy of the result set of a SemQL-to-SPARQL translation executed on an RDF Knowledge Graph that contains the exact same data as the SQL database, the possibility that an incorrect SPARQL query would randomly produce the same result set as the SQL query is very low.

In short, we compare the execution results of the ground truth SQL statements on the Spider PostgreSQL databases and the execution results of our translated SPARQL queries on (1) the baseline knowledge graphs and (2) the enriched knowledge graphs, i.e., the two different RDF versions of the original relational Spider databases.

6.3. ValueNet4SPARQL on Spider

Below, we discuss the accuracy of the translated SPARQL queries on the two different versions of the converted Spider knowledge graphs. We also include the result set accuracy per database for the deterministic translation of SQL queries from the original system ValueNet.

6.3.1. Results on Baseline Knowledge Graphs

58% of the translated SPARQL queries match the equivalent SQL query ground truth result set, when executed against the baseline knowledge graphs. Figure 10 shows a breakdown of the query accuracy per database against the baseline knowledge graphs in yellow. Because the baseline ontologies (which are used for inferring the connections between classes using Algorithm 2) do not contain any information about the direction of the joins between classes (as discussed in Section 4.2), none of the queries with joins are translated correctly.
6.3.2. Results on Enriched Knowledge Graphs

In the second part of the evaluation, we analyze the query performance against the knowledge graphs that have been improved through the techniques of Knowledge Graph Enrichment we discussed in Section 4.

Figure 10 shows the second evaluation of the SemQL to SPARQL translation (in blue) and the considerable improvement of the accuracy of the queries compared to the results on the baseline knowledge graphs. These results show that KG enrichment improves the accuracy of the SPARQL queries on the enriched knowledge graphs compared to the baseline knowledge graphs by 36% for a total accuracy of 94%. The remaining 6% of queries fail for other reasons that we discuss in Section 6.3.4.

6.3.3. Comparing ValueNet and ValueNet4SPARQL

ValueNet4SPARQL outperforms ValueNet in terms of result set accuracy by 4% over all queries. This improvement is shown in Figure 10 where ValueNet4-SPARQL has a higher accuracy than ValueNet in 6 of the databases. In the majority of the data sets, this improvement is due to the new method for inferring GROUP BY-clauses as shown in Algorithm 3.

Figure 10 shows that the largest discrepancy in performance between ValueNet4SPARQL and ValueNet occurs in the voter_1 and flight_2 databases. As described in Section 5.2.1, both systems infer joins using metadata from each data set, i.e., ValueNet4SPARQL uses an ontology to infer the relationship and the direction of a connection between classes, while ValueNet uses the primary / foreign key relationships from the relational database schema to infer joins. ValueNet4SPARQL outperforms ValueNet in the voter_1 and flight_2 databases because its enriched knowledge graph resulting from query-driven generation contains the relationship, which is missing in the relational database schema.
For the two databases students_transcripts_tracking and network_1 ValueNet slightly outperforms ValueNet4SPARQL. We explain the errors of ValueNet4SPARQL in the next section.

6.3.4. Error Analysis of Failed Queries

We now perform a detailed error analysis of the 6% SPARQL queries that do not produce accurate result sets when executed against the enriched knowledge graphs. Figure 11 shows the number of queries with incorrect result sets, based on their difficulty rating. The lowest number of queries that return the correct result sets are in the hard and extra hard categories.

![Fig. 11. This figure shows the accuracy of the SPARQL queries executed on the enriched knowledge graphs according to their difficulty ranking.](https://yale-lily.github.io//spider)

The following items explain the execution accuracy discrepancies between ValueNet and ValueNet4SPARQL.

**Handling OPTIONAL-keyword**: In SPARQL, it is possible to return results with missing data with the OPTIONAL keyword, but ValueNet4SPARQL does not currently support this.

**Handling certain aggregation queries**: Queries with aggregations over certain data types, such as dateTime, although correctly translated, do not return the correct result sets, because SPARQL does not allow this kind of operation.

**Handling certain superlatives**: Superlative queries that have an ORDER BY-clause with a LIMIT higher than 1 are not handled by the original system ValueNet and are therefore also not handled by ValueNet4SPARQL.

---

11See ‘Data Examples’ at https://yale-lily.github.io//spider

12https://downloads.dbpedia.org/2016-04/
Let us discuss the transformations in detail. In Line 8 we added `dbo:Place` which is the derived parent class of the URI `<http://dbpedia.org/resource/Channel_District>`\(^\text{13}\) By adding the parent class of the resource URI, we can then filter it with the resource URI. Because this structure is similar to filtering a column of a table with a value in a SQL query, we are now able to parse this query into SemQL.

As described in Section 3.3, ValueNet4SPARQL relies on the domain and range attributes in order to infer connections between classes and properties in the graph. However, even in a curated ontology such as DBpedia this is not the case. There are many queries in the LC-QuAD 1.0 data set that use triples with properties and classes that are not linked in the DBpedia knowledge graph. Therefore, in transforming the queries, it was necessary to link triples with the ubiquitous owl:Thing class (see Line 9 in the example above).

### 6.4.2. Handling Very Large Ontologies

ValueNet4SPARQL relies on ontology information in both the training and inference modules of the pipeline. The classes and properties are included as input tokens for the transformer model, which can accept a maximum of 2048 tokens\(^\text{14}\). None of the relational database schemas which have been used with ValueNet exceeded this number of input tokens. For instance, the relational database schemas of the Spider data set contain a maximum of 27 tables and an average of 7 columns per table. Ontologies, however, typically contain significantly more classes and properties than an average database schema. The DBpedia 2016-04 ontology, which we used for our experiments, contains 760 classes and 1,105 properties. This already far exceeds the number of tokens which the transformer architecture of ValueNet4SPARQL can accept as input.

In place of using the entire DBpedia ontology, ontological information is inferred from the values found during the NER-preprocessing step of the pipeline. Hence, we used a smarter strategy and only consider the portion of the ontology explicitly involved in each single query. Starting from a value, it is possible to extract the parent class and properties associated with a given value from the knowledge graph.

### 6.4.3. LC-QuAD 1.0 Query Difficulty

Table 2 shows the various query characteristics that contribute to the hardness and complexity of the LC-QuAD data set compared with the Spider data set.

The table clearly shows that the diversity and “hardness” of the LC-QuAD 1.0 data set is not comparable to that of the Spider data set. However, the major challenge of the LC-QuAD data set is the large ontology which is manually curated and often lacks a well-defined schema.

### 6.4.4. Results

We trained ValueNet4SPARQL using the train and test splits (4,000 train and 1,000 test NL/SPARQL query pairs) provided by the creators of the LC-QuAD 1.0 data set. We compare our system against two state-of-the-art systems using the following evaluation metric:

\[
\text{Precision} = \frac{\text{number of correct queries generated}}{\text{number of queries processed by the system}}
\]

\[
\text{Recall} = \frac{\text{number of correct queries generated}}{\text{number of ground truth queries}}
\]

\[
F1 = 2 \times \frac{\text{recall} \times \text{precision}}{\text{recall} + \text{precision}}
\]
Our results show that ValueNet4SPARQL achieves an F1-score of 85% on the LC-QuAD 1.0 data set. As we can see in Table 3, this is an improvement of 17% w.r.t. the state-of-the-art. It is also important to note that both systems report their performance only on a subset of 1,000 test queries. For instance, NSQA only used 200 out of the 1,000 queries for evaluation, while we evaluated our system on the complete test set.

### 6.4.5. Error Analysis of Failed Queries

Slightly less than 5% (48) queries have an error where the model fails to predict the correct property and class. This type of error also occurs in the relational version ValueNet. The main reason for this is when the columns in different tables have similar names and are hard to disambiguate from one another. An example of this are generic column names like id or name that are found frequently in databases, usually across multiple tables [8].

4% (36) of the queries fail due to incorrect value selection. This can occur due to mispellings in the natural language question or the value in the natural language question being abbreviated or lengthened compared to the resource URI. Consider the following natural language question and the corresponding SPARQL-query: Who owns the newspaper which was founded by Nehru?

```
Query: SELECT DISTINCT ?uri WHERE {
  ?t1 a <http://dbpedia.org/ontology/Agent> .
  ?t1 a <http://dbpedia.org/ontology/City> .
  Filter (?t1=<http://dbpedia.org/resource/St._Louis>) .
  Filter (?t1=<http://dbpedia.org/resource/Madison_River>) .
  Filter (?t1=<http://dbpedia.org/resource/Jawaharlal_Nehru>) .
}
```

The example above illustrates a case where string similarity will not be able to find any of the values in the natural language question as neither "St. Louis" nor "Madison River" are present in the natural language question. The number of queries that fail due to the NER failure to find the values could be reduced by adding additional NER heuristics such as semantic similarity.

2% (21) of the queries from the LC-QuAD 1.0 test set fail because the resources in the queries have not been assigned any parent class (not even owl:Thing). This issue could be resolved by using inferencing.

False Negatives: 1% of the errors are due to false negatives. These errors are due to a lowercase value being selected over an uppercase value. This can occur when the value is found as an exact match in the NL question and also in the NER preprocessing.

### 7. Related Work

In this section we revise the related work on (1) knowledge graph construction and enrichment as well as (2) querying databases and knowledge graphs in natural language.

#### 7.1. Knowledge Graph Construction and Enrichment

Several approaches and tools have been proposed to address the problem of automatic KG construction starting from a relational database. The majority of the services rather than exact matches found directly in the NL question. However, for this data set values with exact matching between the candidate values and the NL question performed best for the majority of the queries.

A further 3% (28) of queries fail because the values in the resource URIs cannot be found in the natural language question. The pre-processing pipeline of ValueNet4SPARQL currently uses string similarity to find the values of the resource URIs. Consider the following natural language question and the corresponding SPARQL-query: “Which university with athletics department in NCAA Division I Football Bowl Subdivision has nickname Tulane Green Wave?”

```
Query: SELECT DISTINCT ?uri WHERE {
  ?uri a owl:Thing .
  Filter (?t1=<http://dbpedia.org/resource/St._Louis>) .
  Filter (?t1=<http://dbpedia.org/resource/Madison_River>) .
  Filter (?t1=<http://dbpedia.org/resource/Jawaharlal_Nehru>) .
}
```

In the pre-processing step 2 values are found, “Nehru” because it is a capitalized word in the natural language question and “jawaharlal nehru” from the combined NER services (Google NER\(^1\) and DBpedia Spotlight\(^2\)) built into the preprocessing pipeline. The model incorrectly chooses “Nehru” as the value. This kind of incorrect value prediction could be mitigated by prioritizing matches found through the NER

---

\(^1\)https://cloud.google.com/natural-language
\(^2\)https://www.dbpedia-spotlight.org/
approaches extend the Direct Mapping recommendation, and follow the ODM (Ontop Direct Mapping) idea of producing an R2RML specification which adheres to the Direct Mapping standard. BootOX [22] supports all three OWL 2 QL profiles, and like our approach and ODM generates R2RML mappings. BootOX offers also a set of advanced features, such as exploiting data information like the selectivity of join operators. To the best of our knowledge, BootOX was the first system able to enrich the KG through suitable domain and range OWL 2 QL axioms.

Mirror [12], follows a different strategy: it encodes the datatype information directly in the R2RML mapping, rather than in an OWL ontology. Conceptually, the methodology adopted in Mirror is very close to the one of MPBoot since both systems rely on “patterns” derived from well-established techniques in the fields of conceptual modeling and database design. However, the catalogs of patterns on which the two systems are built, differ for certain cases.

Ultrawrap Mapper [30] is a commercial tool which helps the user in specifying R2RML mappings. Like MPBoot, it allows the user to specify the specific attributes and tables for which the mapping generation should take place. It also offers advanced features such as the possibility of uploading a target domain-ontology for which specific mapping recommendations will be provided based on ontology matching techniques.

A more ambitious system is Karma [18], which in addition to the tools mentioned so far, also supports non-relational and semi-structured sources such as CSV files. Karma does not support a fully automatic mapping generation, but rather supports the user with a sophisticated interface which suggests relationships among the schema elements of a source.

The main advantage that MPBoot brings to the already available tools is the possibility of dealing with poorly structured DB schemas without the need for human intervention: relationships between tables are discovered automatically through a workload analysis of user queries, and added to the ontology in the form of ontology axioms together with their relative domain and range information.

7.2. Querying Databases and Knowledge Graphs in Natural Language

A good overview on the state of the art of querying relational databases in natural language is given in [1, 4, 6, 25, 29]. The main goal of these systems is to automatically translate natural language to SQL. Recent work uses neural machine translation techniques based on transformer architecture [8].

The state of the art algorithms for querying knowledge graphs are described in [14, 15, 26, 31, 34, 37]. While Bio-SODA [31, 32] uses a pattern-based approach for translating natural language to SPARQL without requiring training data, the other two approaches use neural machine translation techniques that require typically large amounts of training data to query DBpedia data sets.

One such data set is LC-Quad 2.0 [16], which is comprised of 30,000 NL/SPARQL query pairs. This data set was generated using 22 different SPARQL query templates over the DBpedia and Wikidata knowledge bases. Although this data set is larger than the Spider data set, it lacks some of the diversity and complexity of the Spider data set, i.e., LC-Quad 2.0 does not contain queries with GROUP BY, HAVING, most aggregations (SUM, MIN, MAX, AVG), nested queries or set operations [13]. Additionally, there have not yet been any submissions to the LC-Quad 2.0 Leader board17, so it is difficult to assess the effectiveness of training systems using this data set.

ValueNet4SPARQL, the approach described in this paper, has a fundamental difference to the above-described approaches since it does not query "pure" relational databases or knowledge graphs. ValueNet4SPARQL is designed for enabling natural language queries for the ontology-based data access paradigm. This allows us to leverage the vast amounts of training data available for relational databases to query knowledge graphs.

8. Conclusions

In this paper, we presented ValueNet4SPARQL, an end-to-end NL-to-SPARQL framework for retrieving high-quality results in ontology-based data access scenarios. Compared to state-of-the-art NL-to-SQL systems, we make three key contributions. First, we show how to adapt neural models that were trained on SQL databases to our context without having the need for any additional training data such as NL/SPARQL pairs. In addition, we show how to enrich the KG layer used for data access by automatically detecting and

17http://lc-quad.sda.tech/
correcting data modelling issues by taking advantages of both workload and instance data. Finally, we introduce a new set of algorithms to faithfully translate complex operations (such as joins or aggregates) into SPARQL, hence generating more accurate results.

We applied our methodology to the Spider data set, thus obtaining a new benchmarking data set for NL-to-SPARQL systems. Our empirical evaluation shows that our approach considerably improves the accuracy of the results—by up to 36% (for a total accuracy of 94%)—compared to a standard baseline. We also evaluated ValueNet4SPARQL on the well known LCQuAD 1.0 data set and achieve an F1-score of 85%, which outperforms the state-of-the-art system by 17%.

As future work, we plan to study the impact of transfer learning with ValueNet4SPARQL on other data sets. We also plan to further develop the system in the context of ontology-based data access for real-world databases used in specialized fields with vocabularies that are highly divergent from the more generalized vocabulary of the majority of NL-to-SPARQL or NL-to-SQL data sets.

Acknowledgments

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No. 863410.

References


