RDFFrames: Knowledge Graph Access for Machine Learning Tools

Aisha Mohamed[★*] Ghadeer Abuoda[♦*] Abdurrahman Ghanem[▲†]
Zoi Kaoudi[▼†] Ashraf Aboulnaga[★]
[★]Qatar Computing Research Institute, HBKU [♦]College of Science and Engineering, HBKU
[▲]Bluescape [▼]Technische Universität Berlin

ABSTRACT

Knowledge graphs represented in RDF are becoming increasingly popular and are essential to many machine learning applications. A rich ecosystem of RDF data management systems and tools has evolved over the years, most notably RDF database management systems that support the SPARQL query language. Surprisingly, machine learning tools for knowledge graphs typically do not use SPARQL despite the obvious advantages of using a database system. This is due to the mismatch between SPARQL and machine learning tools in terms of expected data model and interface style. Machine learning tools work on data in tabular format and process it using imperative relational API calls, while SPARQL matches graph patterns to RDF triples. To access knowledge graphs for machine learning, we observe that it is more natural to use a navigational paradigm based on graph traversal rather than the SPARQL paradigm based on triple patterns. We demonstrate RDFFrames, a framework that bridges the gap between machine learning tools and RDF database systems by offering the usability and flexibility of machine learning tools together with the performance of a database system. RDFFrames enables the user to make a sequence of Python calls to define the data to be extracted from a knowledge graph stored in an RDF database system, and it translates these calls into a compact SPARQL query, executes it on the database system, and returns the results in a standard tabular format.

PVLDB Reference Format:
Aisha Mohamed, Ghadeer Abuoda, Abdurrahman Ghanem, Zoi Kaoudi, and Ashraf Aboulnaga. RDFFrames: Knowledge Graph Access for Machine Learning Tools. *PVLDB*, 13(12): 2889 - 2892, 2020.
DOI: https://doi.org/10.14778/3415478.3415501

[*]Joint first authors.
[†]Work done while at QCRI.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415501

1. INTRODUCTION

There has recently been a sharp growth in the number of knowledge graph datasets that are made available in the RDF (Resource Description Framework)[1] data model. Examples include general knowledge graphs such as DBpedia, YAGO, and Wikidata, and domain-specific knowledge graphs such as BioRDF. The rich information and semantic structure of knowledge graphs makes them useful in many machine learning applications such as recommender systems, search, and question answering. Recently, many machine learning algorithms have been developed specifically for analyzing knowledge graphs (e.g., [3]).

The RDF data model provides a powerful abstraction for representing heterogeneous, incomplete, and potentially noisy knowledge graphs. A rich ecosystem of data management systems and tools that support RDF has evolved over the years. This ecosystem includes standard serialization formats, parsing and processing libraries, and most notably RDF database management systems (a.k.a. *RDF engines* or *triple stores*) that support SPARQL, the W3C standard query language for RDF data. Examples of these systems include Virtuoso,[2] Apache Jena, and managed services such as Amazon Neptune.

We make the observation that none of the publicly available machine learning tools for knowledge graphs that we are aware of uses SPARQL and RDF engines. This, despite the obvious advantage of using a database system such as data independence, declarative querying, and efficient and scalable query processing. For example, we investigated all the prominent open-source implementations of knowledge graph embeddings, an active research area, and we found that they process data by ad-hoc scripts rather than using SPARQL.

We posit that machine learning tools do not use RDF engines due to an "impedance mismatch." Specifically, machine learning software stacks are based on data in *tabular format* and the split-apply-combine paradigm [4]. An example tabular format is the highly popular *dataframes*, supported by libraries in several languages such as Python and R, and by systems such as Apache Spark. Thus, the first step in most machine learning pipelines is to identify the required data and extract this data into a table. We observe that it is more natural in machine learning to identify and extract data from a knowledge graph into a table using *navigation* in the graph, rather than the declarative, pattern-based querying provided by SPARQL.

[1]`https://www.w3.org/RDF`
[2]`https://virtuoso.openlinksw.com`

In this demonstration, we showcase *RDFFrames*, a framework that bridges the gap between machine learning tools and RDF engines. Specifically, RDFFrames provides a set of operators that use the navigational paradigm familiar in machine learning to explore an RDF graph, identify the data required from this graph, and extract this data into a tabular format. RDFFrames processes the operators called by the user and produces a corresponding SPARQL query, executes it on an RDF engine or SPARQL endpoint, and returns the results as a table. In principle, the RDFFrames operators can be implemented in any programming language and can return data in any tabular format. However, concretely, our current implementation of RDFFrames is a Python library that returns data as dataframes of the popular pandas library[3] so that further processing can be done in the rich PyData ecosystem. RDFFrames is available as open source[4] and via the Python pip installer.

The novelty of RDFFrames lies mainly in two aspects: First, the API provided to the user is designed to be intuitive and flexible. The API consists of navigational operators and data processing operators based on familiar relational algebra operations. Second, RDFFrames retrieves the required data efficiently by using lazy execution and converting the operators called by a user into compact and efficient SPARQL. We present an overview of RDFFrames in Section 2. A full description can be found in [1].

The demonstration is based on machine learning applications that use RDFFrames to access data in real knowledge graphs. Demonstration participants can interact with these applications, visualize the input graph, observe and change the RDFFrames calls used to retrieve data, observe and change the corresponding SPARQL, execute it against an RDF engine, and interactively run the machine learning task. In general, data access for machine learning is an important topic, and the demonstration explores this topic for RDF knowledge graphs.

2. OVERVIEW OF RDFFrames

This section provides an overview of RDFFrames (more details in [1]). We describe the Python API then discuss how API calls are converted to SPARQL.

Python API: The goal of the API is to allow users to construct a tabular dataset from a knowledge graph for a specific machine learning task. The API provides functions to explore a knowledge graph and initialize the tabular dataset with relevant entities (RDF resources), expand a dataset by following RDF predicates (edges in the knowledge graph), filter the extracted data according to simple or complex conditions, group and aggregate the extracted data, and combine datasets through the join operator. The challenge when designing this API was to provide a set of functions that enable complex query operations (e.g., group-by and join) and can be composed with each other in a flexible way (e.g., applying a filter on the results of group-by and aggregation), all while maintaining the navigational nature of the API.

The core Python class of the API is the `Dataset` class. This class is a logical representation of a tabular dataset that is retrieved from an RDF graph. The user calls the functions of this class to create a logical description of the data that she wants. It is important to note that creating this logical description *does not cause queries to be generated or executed*. This is done later using lazy execution.

[3] `https://pandas.pydata.org`

[4] `https://github.com/qcri/rdfframes`

```
data = graph.feature_domain_range(
    "dbp:starring", "movie", "actor")
american = data.expand(
        "actor",[("dbp:birthPlace", "country")])
    .filter({"country": ["=dbpr:UnitedStates"]})
prolific = data.group_by(["actor"])
    .count("movie", "movie_count")
    .filter({"movie_count": [">=20"]})
actors = american.join(prolific,"actor", OuterJoin)
```

Listing 1: RDFFrames code for American or prolific actors and their movies.

Example code using the `Dataset` class is shown in Listing 1. This code is a succinct version of the first application in the demo scenarios of Section 3, and it identifies actors and the movies they have starred in. The returned data is restricted to actors who are American or prolific (defined as having 20 or more movies). The core functions of the `Dataset` class can be grouped into the following categories:

- Initialization: These are functions that logically retrieve columns from an RDF knowledge graph to initialize the `Dataset` based on a specific condition (e.g., instances of a specific class or pairs of entities connected by a predicate). The `feature_domain_range` function called in the first line of Listing 1 is an example initialization function. This function logically creates a table with two columns, `movie` and `actor`, containing all pairs of RDF entities linked by a `starring` RDF predicate, that is, all movies and actors starring in them. We reiterate that this call does not access the RDF engine yet, but rather provides a logical description of the desired dataset.
- Expand: The main operation in RDFFrames is navigating the edges of the knowledge graph. This operation is implemented in the `expand` function. This function adds columns to a dataset by "expanding" entities in an existing source column, that is, traversing edges in the graph starting from these entities. The first argument to this function is the source column. The second argument is a list of (`predicate`, `new_col`) pairs. The function follows the edges representing `predicate` from/to entities in the source column and adds the resources reached by this navigation to a new column named `new_col`. If there is no edge in the graph to expand an existing row in the dataset, the function can either drop that row (inner-join semantics) or keep it but add a null value to the new column (outer-join semantics). In Listing 1, `expand` is used to add each actor's country to the dataset.
- Relational Operators: The `Dataset` class provides functions for relational operators such as filtering, grouping, aggregation, and join. These functions are used to manipulate a dataset (or join two datasets) using familiar relational semantics. Note that even though these functions logically view the dataset as a table, RDFFrames translates them to SPARQL. Relational operators can be implemented in pandas when the dataframe is returned, but including them in the RDFFrames API increases efficiency since they can be pushed into the RDF engine [1].

In addition to these core functions, RDFFrames provides other functions for, e.g., exploring the RDF data, defining namespaces, handling the connection and communication with the RDF engine, caching a dataset, and others.

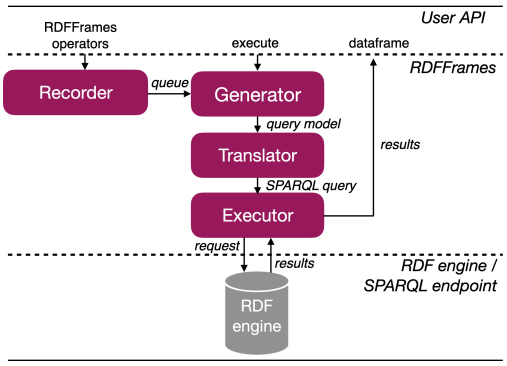

Figure 1: RDFFrames architecture.

Conversion to SPARQL: A naive way to implement RDFFrames is to convert every API call to a SPARQL query and eagerly send these queries to the RDF engine. This would be highly inefficient, as shown in [1]. Instead, `Dataset` objects are designed to be logical descriptions of a dataset, and RDFFrames generates SPARQL queries to physically create the dataset using the architecture presented in Figure 1.

The sequence of function calls to a `Dataset` object is recorded and then used to generate an intermediate representation of the required dataset termed a *query model*. The goal of the query model is (i) to separate the API parsing logic from the query construction logic for flexible implementation, and (ii) to simplify optimizing the constructed SPARQL query, especially in the case of nested queries. Inspired by the Query Graph Model [2], the query model is a structure that contains, among other information, returned columns, graph patterns, filter conditions, and references to other query models in the case of nested queries [1].

After the query model is created, a translation algorithm generates the corresponding SPARQL query. Our design of the query model and translation algorithm guarantees that any valid sequence of function calls to a `Dataset` can be converted to a query model, and any query model can be translated to a single, semantically equivalent SPARQL query. This SPARQL query is created and sent to the RDF engine when the user calls a special `execute` function, and the results are returned as a pandas dataframe.

The SPARQL query corresponding to the code in Listing 1 is shown in Listing 2. *We believe that the RDFFrames code in Listing 1 is simpler than the corresponding SPARQL query in Listing 2, in addition to being better suited to the Python machine learning environment and better integrated with this environment. This is the primary motivation behind RDFFrames.* Note that this query is certainly not the most complex that we have seen in our experience with RDFFrames.

Several features of RDFFrames make it efficient and convenient to use (the performance of RDFFrames is studied in [1]). First, RDFFrames adopts a lazy execution model, generating and processing a query only when needed. Second, RDFFrames always generates exactly one SPARQL query for each dataset, never more. This minimizes the number of interactions with the RDF engine and gives the query optimizer a chance to explore all optimization opportunities. Third, RDFFrames generates a SPARQL query that is as compact and simple as possible by minimizing the use of nested subqueries and union operators in SPARQL, since

```
SELECT ?actor FROM <http://dbpedia.org> WHERE
 { { SELECT * WHERE
      { { SELECT * WHERE
           { ?movie  dbpp:starring    ?actor .
             ?actor  dbpp:birthPlace  ?country
             FILTER (?country = dbpr:UnitedStates)
           }
        }
        OPTIONAL
        { { SELECT ?actor (COUNT(DISTINCT ?movie)
            AS ?movie_count)
           WHERE
            { ?movie  dbpp:starring  ?actor }
           GROUP BY ?actor
           HAVING (COUNT(DISTINCT ?movie) >= 20)
          }
        }
      }
   }
   UNION
   { SELECT * WHERE
      { { SELECT ?actor (COUNT(DISTINCT ?movie)
          AS ?movie_count)
         WHERE
          { ?movie  dbpp:starring  ?actor }
         GROUP BY ?actor
         HAVING (COUNT(DISTINCT ?movie) >= 20)
        }
        OPTIONAL
        { { SELECT * WHERE
             { ?movie  dbpp:starring    ?actor .
               ?actor  dbpp:birthPlace  ?country
               FILTER (?country = dbpr:UnitedStates)
             }
          }
        }
      }
   }
 }
```

Listing 2: SPARQL query corresponding to Listing 1.

these are known to be expensive. Finally, RDFFrames handles the mechanics of query processing such as connecting to the RDF engine (or SPARQL endpoint) and pagination (i.e., retrieving the results in chunks) to avoid timeouts.

3. DEMONSTRATION SCENARIOS

Our demonstration scenarios provide an interactive experience showcasing the usability, flexibility, and efficiency of using RDFFrames to process knowledge graphs and the advantage of integrating RDFFrames in the PyData ecosystem. Demonstration participants interact with RDFFrames through a Jupyter notebook interface. The demonstration uses multiple knowledge graphs, such as DBpedia, YAGO3, and DBLP, stored on a local instance of the Virtuoso RDF engine. Several machine learning applications that use RDFFrames are provided as part of the demonstration. These applications allow demonstration participants to interactively explore the steps involved in building a pandas dataframe from a knowledge graph using RDFFrames. We describe the interactive elements of our demonstration next, followed by an overview of three of the applications used.

Initially, a demonstration participant chooses a knowledge graph and an application. The participant can visualize the knowledge graph and focus on the part of it that will be processed by RDFFrames. An example of this is shown in Figure 2. In a real usage of RDFFrames, this visualization would help the user write her RDFFrames code. We do not expect demonstration participants to write RDFFrames code or SPARQL queries (although they can do that if they want). Still, the visualization helps the participant better understand the code that we provide.

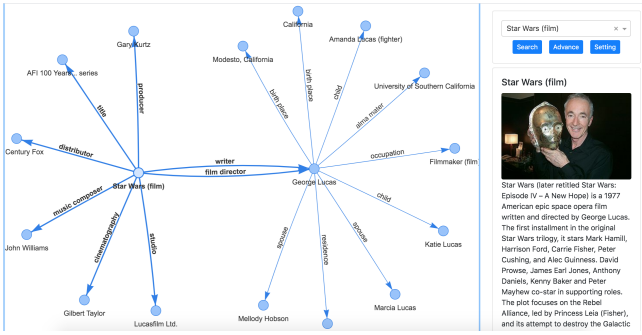

Figure 2: Visualization of part of the DBpedia graph.

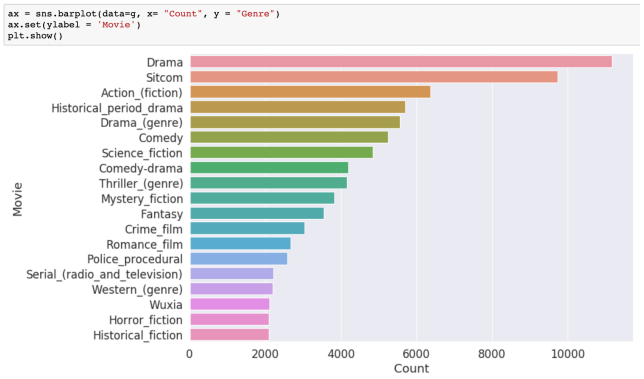

Figure 3: Most occurring genres in the dataframe of movies in DBpedia returned by Listing 1.

The demonstration participant can interactively edit and execute the provided RDFFrames code. Similarly, the participant can edit the generated SPARQL queries and execute them directly on the RDF engine. The ability to see the effect of edits to code and queries can provide insights into the operation of RDFFrames.

After the RDFFrames code is executed, the participant can explore the dataframe returned by RDFFrames and run standard machine learning tasks on it using off-the-shelf tools from the PyData ecosystem (e.g., scikit-learn). The demonstration provides visualizations for the inputs and outputs of the applications. For example, Figure 3 shows a visualization of the most occurring movie genres in the dataframe returned by the code in Listing 1. Note that RDFFrames is concerned with data access and preparation, so the specific machine learning task and its visualization are tangential to its operation. However, running and visualizing these tasks helps the user better understand how RDFFrames fits in the end-to-end machine learning pipeline.

3.1 Demonstration Applications

Movie Genre Classification: Classification is a basic supervised machine learning task. Many knowledge graphs, such as DBpedia and YAGO3, are heterogeneous and contain diverse general information about different topics, so extracting a topic-focused dataframe for a classification task is challenging. In this application, we use RDFFrames to build a dataframe of movies from a knowledge graph, along with a set of movie attributes that can be used for movie genre classification. An expanded version of the RDFFrames code shown in Listing 1 is used in this application. The code performs the following steps to extract a movies dataset from DBpedia: We identify movies that star American actors, since they are assumed to have a global reach. We also identify movies that star prolific actors, defined as those with 20 or more movies. We then build a dataset of movies starring an American or prolific actor and return, for each movie, the genre whenever it is available (it is an optional RDF predicate) and a set of attributes that can be used to predict the genre. This final dataframe is considered a good representative of the global movie industry and can be used to train a classifier to predict the genre for movies that do not have one, using any standard PyData classification algorithm.

Topic Modeling: Topic modeling is a statistical technique commonly used to identify hidden contextual topics in text. In this application, we show the usage of RDFFrames to query the DBLP bibliography dataset in RDF format for conducting topic modeling to identify the active areas of database research. The dataframe required for this task is extracted from the DBLP knowledge graph through a sequence of RDFFrames operators as follows: First, we identify the authors who have published more than 20 papers in SIGMOD and VLDB since the year 2000, which requires using the RDFFrames grouping, aggregation, and filtering capabilities. For the purpose of this application, these are considered the thought leaders in databases. Next, we find the titles of all papers published by these authors since 2010. We then run topic modeling on these titles to identify active areas of research. This application shows the benefit of RDFFrames producing its output as a pandas dataframe, since it utilizes the rich PyData ecosystem (NLP libraries for stop-word removal and scikit-learn for topic modeling).

Knowledge Graph Exploration: One of the main challenges of exposing knowledge graphs to machine learning tools is the complex, heterogeneous structure of the graphs. Thus, the first step for a data scientist working with any unfamiliar knowledge graph is exploring this graph to identify the classes, instances, features of instances in each class, data distributions, and other statistical properties of the graph. RDFFrames provides several convenience functions to support this exploration, and this application showcases these functions. For example, the demonstration participants can get all the classes in the graph and the number of instances of each class. They can sort the classes by their frequency to see the main classes. For each class, they can get the features of instances of this class and the distribution of the features, and a table of the instances of the class with all features or with a subset of the features. This exploration is helpful in understanding the underlying structure of the graph and is the starting point of most processing tasks.

4. REFERENCES

[1] A. Mohamed, G. Abuoda, Z. Kaoudi, A. Ghanem, and A. Aboulnaga. RDFFrames: Knowledge graph access for machine learning tools. *arXiv*, 2002.03614, 2020.

[2] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *SIGMOD*, 1992.

[3] Y. Wang, R. Gemulla, and H. Li. On multi-relational link prediction with bilinear models. In *AAAI*, 2018.

[4] H. Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40:1–29, 2011.