

RDFFrames: Knowledge Graph Access for Machine Learning Tools

Aisha Mohamed¹ · Ghadeer Abuoda¹ · Abdurrahman Ghanem² · Zoi Kaoudi² · Ashraf Abounaga

Received: date / Accepted: date

Abstract Knowledge graphs represented as RDF datasets are integral to many machine learning applications. RDF is supported by a rich ecosystem of data management systems and tools, most notably RDF database systems that provide a SPARQL query interface. Surprisingly, machine learning tools for knowledge graphs 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 data model and programming style. Machine learning tools work on data in tabular format and process it using an imperative programming style, while SPARQL is declarative and has as its basic operation matching graph patterns to RDF triples. We posit that a good interface to knowledge graphs from a machine learning software stack should use an imperative, navigational programming paradigm based on graph traversal rather than the SPARQL query paradigm based on graph patterns. In this paper, we present RDFFrames, a

framework that provides such an interface. RDFFrames provides an imperative Python API that gets internally translated to SPARQL, and it is integrated with the PyData machine learning software stack. 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 SPQARL query, executes it on the database system, and returns the results in a standard tabular format. Thus, RDFFrames is a useful tool for data preparation that combines the usability of PyData with the flexibility and performance of RDF database systems.

Keywords Knowledge Graphs · RDF · SPARQL · PyData · Data Preparation · Machine Learning

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 knowledge graphs that cover a broad set of domains such as DBpedia [25], YAGO [41], Wikidata [42], and BabelNet [29], as well as specialized graphs for specific domains like product graphs for e-commerce [13], biomedical information networks [6], and bibliographic datasets [16, 27]. The rich information and semantic structure of knowledge graphs makes them useful in many machine learning applications [10], such as recommender systems [21], virtual assistants, and question answering systems [45]. Recently, many machine learning algorithms have been developed specifically for knowledge graphs, especially

A. Mohamed
Qatar Computing Research Institute, HBKU
E-mail: ahmohamed@hbku.edu.qa

G. Abuoda
College of Science and Engineering, HBKU
E-mail: gabuoda@hbku.edu.qa

G. Ghanem
Bluescape
E-mail: ghanemabdo@gmail.com

Z. Kaoudi
Technische Universität Berlin
E-mail: zoi.kaoudi@tu-berlin.de

A. Abounaga
Qatar Computing Research Institute, HBKU
E-mail: aabounaga@hbku.edu.qa

¹Joint first authors.

²Work done while at QCRI.

¹ <https://www.w3.org/RDF>

in the sub-field of *relational learning*, which is dedicated to learning from the relations between entities in a knowledge graph [30, 32, 44].

RDF is widely used to publish knowledge graphs as it provides a powerful abstraction for representing heterogeneous, incomplete, sparse, and potentially noisy knowledge graphs. RDF is supported by a rich ecosystem of data management systems and tools that 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,² the W3C standard query language for RDF data. Examples of these systems include OpenLink Virtuoso,³ Apache Jena,⁴ and managed services such as Amazon Neptune.⁵

However, we make the observation that none of the publicly available machine learning or relational learning tools for knowledge graphs that we are aware of uses SPARQL to explore and extract datasets from knowledge graphs stored in RDF database systems. 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 recent open source relational learning implementations, and we found that they all rely on ad-hoc scripts to process very small knowledge graphs and prepare the necessary datasets for learning. This observation applies to the implementations of published state-of-the-art embedding models, e.g., scikit-kge [31, 33],⁶ and also holds for the recent Python libraries that are currently used as standard implementations for training and benchmarking knowledge graph embeddings, e.g., Ampligraph [8], OpenKE [20], and PyKEEN [3]. These scripts are limited in performance, which slows down data preparation and leaves the challenges of applying embedding models on the scale of real knowledge graphs unexplored.

We posit that machine learning tools do not use RDF engines due to an “impedance mismatch.” Specifically, typical machine learning software stacks are based on data in *tabular format* and the split-apply-combine paradigm [46]. An example tabular format is the highly popular *dataframes*, supported by libraries in several languages such as Python and R (e.g., the pandas⁷ and scikit-learn libraries in Python), and by systems such as Apache Spark [47]. Thus, the first step in most ma-

chine learning pipelines (including relational learning) is a data preparation step that explores the knowledge graph, identifies the required data, extracts this data from the graph, efficiently processes and cleans the extracted data, and returns it in a table. Identifying and extracting this refined data from a knowledge graph requires efficient and flexible *graph traversal* functionality. SPARQL is a declarative pattern matching query language designed for distributed data integration with unique identifiers rather than navigation [26]. Hence, while SPARQL has the expressive power to process and extract data into tables, machine learning tools do not use it since it lacks the required flexibility and ease of use of *navigational* interfaces.

In this paper, we introduce *RDFFrames*, a framework that bridges the gap between machine learning tools and RDF engines. RDFFrames is designed to support the data preparation step. It defines a user API consisting of two type of operators: navigational operators that explore an RDF graph and extract data from it based on a graph traversal paradigm, and relational operators for processing this data into refined clean datasets for machine learning applications. The sequence of operators called by the user represents a logical description of the required dataset. RDFFrames translates this description to a SPARQL query, executes it on an RDF engine, 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 so that further processing can leverage the richness of the PyData ecosystem. RDFFrames is available as open source⁸ and via the Python pip installer. It is implemented in 6,525 lines of code, and was demonstrated in [28].

Motivating Example. We illustrate the end-to-end operation of RDFFrames through an example. Assume the DBpedia knowledge graph is stored in an RDF engine, and consider a machine learning practitioner who wants use DBpedia to study prolific American actors (defined as those who have starred in 50 or more movies). Let us say that the practitioner wants to see the movies these actors starred in and the Academy Awards won by any of them. Listing 1 shows Python code using the RDFFrames API that prepares a dataframe with the data required for this task. It is important to note that this code is a *logical description* of the dataframe and *does not cause a query to be generated or data to be retrieved from the RDF engine.* At

² <https://www.w3.org/TR/sparql11-query>

³ <https://virtuoso.openlinksw.com>

⁴ <https://jena.apache.org>

⁵ <https://aws.amazon.com/neptune>

⁶ <https://github.com/mnick/scikit-kge>

⁷ <https://pandas.pydata.org>

⁸ <https://github.com/qcri/rdfframes>

the end of a sequence of calls such as these, the user calls a special `execute` function that causes a SPARQL query to be generated and executed on the engine, and the results to be returned in a dataframe.

The first statement of the code creates a two-column RDFFrame with the URIs (Universal Resource Identifiers) of all movies and all the actors who starred in them. The second statement navigates from the actor column in this RDFFrame to get the birth place of each actor and uses a filter to keep only American actors. Next, the code finds all American actors who have starred in 50 or more movies (prolific actors). This requires grouping and aggregation, as well as a filter on the aggregate value. The final step is to navigate from the actor column in the prolific actors RDFFrame to get the actor’s Academy Awards (if available). The result dataframe will contain the prolific actors, movies that they starred in, and their Academy Awards if available. An expert-written SPARQL query corresponding to Listing 1 is shown in Listing 2. RDFFrames provides an alternative to writing such a query that is simpler and closer to the navigational paradigm, and is better-integrated with the machine learning environment. The case studies in Section 6.1 describe more complex data preparation tasks and present the RDFFrames code for these tasks and the corresponding SPARQL queries.

```
movies = graph.feature_domain_range('dbp:starring',
    'movie', 'actor')
american = movies.expand('actor',
    [('dbp:birthPlace', 'country')])\
    .filter({'country': ['=dbpr:United_States']})
prolific = american.group_by(['actor'])\
    .count('movie', 'movie_count')\
    .filter({'movie_count': ['>=50']})
result = prolific.expand('actor', [('dbpp:starring',
    'movie', INCOMING), ('dbpp:academyAward', 'award',
    OPTIONAL)])
```

Listing 1: RDFFrames code - Prolific American actors who have Academy Awards.

```
SELECT *
FROM <http://dbpedia.org>
WHERE
{ ?movie dbpp:starring ?actor
  { SELECT DISTINCT ?actor
    (COUNT(DISTINCT ?movie) AS ?movie_count)
    WHERE
    { ?movie dbpp:starring ?actor .
      ?actor dbpp:birthPlace ?actor_country
      FILTER ( ?actor_country =
        dbpr:United_States )
    }
  }
  GROUP BY ?actor
  HAVING ( COUNT(DISTINCT ?movie) >= 50 )
}
OPTIONAL
{ ?actor dbpp:academyAward ?award }
```

Listing 2: Expert-written SPARQL query corresponding to RDFFrames code shown in Listing 1.

RDFFrames in a Nutshell. The architecture of RDFFrames is shown in Figure 1. At the top of the figure is the user API, which consists of a set of operators implemented as Python functions. We make a design

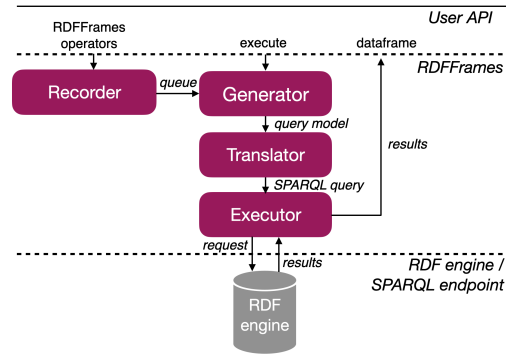


Fig. 1: RDFFrames architecture.

decision in RDFFrames to use a *lazy evaluation strategy*. Thus, the **Recorder** records the operators invoked by the user without executing them, storing the operators in a FIFO queue. The special `execute` operator causes the **Generator** to consume the operators in the queue and build a *query model* representing the user’s code. The query model is an intermediate representation for SPARQL queries. The goal of the query model is (i) to separate the API parsing logic from the query building logic for flexible manipulation and implementation, and (ii) to facilitate optimization techniques for building the queries, especially in the case of nested queries. Next, the **Translator** translates the query model into a SPARQL query. This process includes validation to ensure that the generated query has valid SPARQL syntax and is equivalent to the user’s API calls. Our choice to use lazy evaluation means that the entire sequence of operators called by the user is captured in the query model processed by the **Translator**. We design the **Translator** to take advantage of this fact and generate *compact and efficient* SPARQL queries. Specifically, each query model is translated to one SPARQL query and the **Translator** minimizes the number of nested subqueries. After the **Translator**, the **Executor** sends the generated SPARQL query to an RDF engine or SPARQL endpoint, handles all communication issues, and returns the results to the user in a dataframe. **Contributions.** The novelty of RDFFrames lies in:

- First, the API provided to the user is designed to be intuitive and flexible, in addition to being expressive. The API consists of navigational operators and data processing operators based on familiar relational algebra operations such as filtering, grouping, and joins (Section 3).
- Second, RDFFrames translates the API calls into efficient SPARQL queries. A key element in this is the query model which exposes query equivalences in a simple way. In generating the query model from a sequence of API calls and in generating the SPARQL

query from the query model, RDFFrames has the overarching goal of generating efficient queries (Section 4). We prove the correctness of the translation from API calls to SPARQL. That is, we prove that the dataframe that RDFFrames returns is semantically equivalent to the results set of the generated SPARQL query (Section 5).

- Third, RDFFrames handles all the mechanics of processing the SPARQL query such as the connection to the RDF engine or SPARQL endpoint, pagination (i.e., retrieving the results in chunks) to avoid the endpoint timing out, and converting the result to a dataframe. We present case studies and performance comparisons that validate our design decisions and show that RDFFrames outperforms several alternatives (Section 6).

2 Related Work

Data Preparation for Machine Learning. It has been reported that 80% of data analysis time and effort is spent on the process of exploring, cleaning, and preparing the data [9], and these activities have long been a focus of the database community. For example, the recent Seattle report on database research [1] acknowledges the importance of these activities and the need to support data science ecosystems such as PyData and to “devote more efforts on the end-to-end data-to-insights pipeline.”

This paper attempts to reduce the data preparation effort by defining a powerful API for accessing knowledge graphs. To underscore the importance of such an API, note that [40] makes the observation that most of the code in a machine learning system is devoted to tasks other than learning and prediction. These tasks include collecting and verifying data and preparing it for use in machine learning packages. This requires a massive amount of “glue code”, and [40] observes that this glue code can be eliminated by using well-defined common APIs for data access (such as RDFFrames).

Some related work focuses on the end-to-end machine learning life cycle (e.g., [2, 5, 48]). Some systems, such as MLdp [2], focus primarily on managing input data, but they do not have special support for knowledge graphs. RDFFrames provides such support.

Database Support for PyData. Some recent efforts to provide database support for the PyData ecosystem focus on the scalability of dataframe operations, while other efforts focus on replacing SQL as the traditional data access API with pandas-like APIs. Koalas⁹ implements the pandas dataframe API on top of Apache Spark for better scalability. Modin [35] is a scalable

dataframe system based on a novel formalism for pandas dataframes. Ibis¹⁰ defines a variant of the dataframe API (not pandas) and translates it to SQL so that it can execute on a database system for scalability. Ibis also supports other backends such as Spark. Koalas and Modin do not support SQL backends, and Ibis does not have a pandas API. A recent system that addresses these limitations is Magpie [22], which translates pandas operations to Ibis for scalable execution on multiple backends, including SQL database systems. Magpie chooses the best backend for a given program based on the program’s complexity and the data size. Grizzly [19] is a framework that generates SQL queries from a pandas-like API and ships the SQL to a standard database system for scalable execution. Grizzly relies on the database system’s support for external tables in order to load the data. It also creates user UDFs as native UDFs in the database system. The AIDA framework [14] allows users to write relational and linear algebra operators in Python and pushes the execution of these operators into a relational database system.

All of these recent works are similar in spirit to RDFFrames in that they replace SQL for data access with a pandas-like API and/or rely on a database backend for scalability. However, all of the works focus on relational data and not graph data. RDFFrames is the first to define a pandas-like API for graph data (specifically RDF), and to support a graph database system as a scalable backend.

Why RDF? Knowledge graphs are typically represented in the RDF data model. Another popular data model for graphs is the *property graph* data model, which has labels on nodes and edges as well as (property, value) pairs associated with both. Property graphs have gained wide adoption in many applications and are supported by popular database systems such as Neo4j¹¹ and Amazon Neptune. Multiple query languages exist for property graphs, and efforts are underway to define a common powerful query language [4].

A popular query language for property graphs is Gremlin.¹² Like RDFFrames, Gremlin adopts a navigational approach to querying the graph, and some of the RDFFrames operators are similar to Gremlin operators. The popularity of Gremlin is evidence that a navigational approach is attractive to users. However, all publicly available knowledge graphs including DBpedia [25] and YAGO [38] are represented in RDF format. Converting RDF graphs to property graphs is not straightforward mainly because the property graph

⁹ <https://koalas.readthedocs.io/en/latest>

¹⁰ <https://ibis-project.org>

¹¹ <https://neo4j.com>

¹² <https://tinkerpop.apache.org/gremlin.html>

model does not provide globally unique identifiers and linking capability as a basic construct. In RDF knowledge graphs, each entity and relation is uniquely identified by a URI, and links between graphs are created by using the URIs from one graph in the other. RDFFrames offers a navigational API similar to Gremlin to data scientists working with knowledge graphs in RDF format and facilitates the integration of this API with the data analysis tools of the PyData ecosystem.

Why SPARQL? RDFFrames uses SPARQL as the interface for accessing knowledge graphs. In the early days of RDF, other query languages were proposed (see [18] for a survey), but none of them has seen broad adoption, and SPARQL has emerged as the standard.

Some work proposes navigational extensions to SPARQL (e.g., [24, 34]), but these proposals add complex navigational constructs such as path variables and regular path expressions to the language. In contrast, the navigation used in RDFFrames is simple and well-supported by standard SPARQL without extensions. The goal of RDFFrames is not complex navigation, but rather providing a simple yet rich suite of common data access and preparation operators that can be integrated in a machine learning pipeline.

Python Interfaces. A Python interface for accessing RDF knowledge graphs is provided by Google’s Data Commons project.¹³ However, the goal of that project is not to provide powerful data access, but rather to synthesize a single graph from multiple knowledge graphs, and to enable browsing for graph exploration. The provided Python interface has only one data access primitive: following an edge in the graph in either direction, which is but one of many capabilities in RDFFrames.

The Magellan project [17] provides a set of interoperable Python tools for entity matching pipelines. It is another example of developing data management solutions by extending the PyData ecosystem [12], albeit in a very different domain from RDFFrames. The same factors that made Magellan successful in the world of entity matching can make RDFFrames successful in the world of knowledge graphs.

There are multiple recent Python libraries that provide access to knowledge graphs through a SPARQL endpoint over HTTP. Examples include `pysparql`,¹⁴ `sparql-client`,¹⁵ and `AllegroGraph Python client`.¹⁶ However, all these libraries solve a very different (and simpler) problem compared to RDFFrames: they take

a SPARQL query written by the user and handle sending this query to the endpoint and receiving results. On the other hand, the main focus of RDFFrames is *generating* the SPARQL query from imperative API calls. Communicating with the endpoint is also handled by RDFFrames, but it is not the primary contribution.

Internals of RDFFrames. The internal workings of RDFFrames involve a logical representation of a query. Query optimizers use some form of logical query representation, and we adopt a representation similar to the Query Graph Model [36]. Another RDFFrames task is to generate SPARQL queries from a logical representation. This task is also performed by systems for federated SPARQL query processing (e.g., [39]) when they send a query to a remote site. However, the focus in these systems is on answering SPARQL triple patterns at different sites, so the queries that they generate are simple. RDFFrames requires more complex queries so it cannot use federated SPARQL techniques.

3 RDFFrames API

This section presents an overview of the RDFFrames API. RDFFrames provides the user with a set of operators, where each operator is implemented as a function in a programming language. Currently, this API is implemented in Python, but we describe the RDFFrames operators in generic terms since they can be implemented in any programming language. The goal of RDFFrames is to build a table (the dataframe) from a subset of information extracted from a knowledge graph. We start by describing the data model for a table constructed by RDFFrames, and then present an overview of the API operators.

3.1 Data Model

The main tabular data structure in RDFFrames is called an *RDFFrame*. This is the data structure constructed by API calls (RDFFrames operators). RDFFrames provides initialization operators that a user calls to initialize an RDFFrame and other operators that extend or modify it. Thus, an RDFFrame represents the data described by a sequence of one or more RDFFrames operators. Since RDFFrames operators are not executed on relational tables but are mapped to SPARQL graph patterns, an RDFFrame is not represented as an actual table in memory but rather as an abstract description of a table. A formal definition of a knowledge graph and an RDFFrame is as follows:

Definition 1 (Knowledge Graph) A knowledge graph $G : (V, E)$ is a directed labeled RDF graph where the set of nodes $V \in I \cup L \cup B$ is a set of RDF URIs I , literals L , and blank nodes B existing in G , and the set of labeled edges E is a set of ordered pairs of elements of V having labels from I . Two nodes connected

¹³ <http://datacommons.org>

¹⁴ <https://code.google.com/archive/p/pysparql>

¹⁵ <https://pypi.org/project/sparql-client>

¹⁶ <https://franz.com/agraph/support/documentation/current/python>

by a labeled edge form a *triple* denoting the relationship between the two nodes. The knowledge graph is represented in RDFFrames by a *graph-uri*.

Definition 2 (RDFFrame) Let \mathbb{R} be the set of real numbers, N be an infinite set of strings, and V be the set of RDF URIs and literals. An RDFFrame D is a pair $(\mathcal{C}, \mathcal{R})$, where $\mathcal{C} \subseteq N$ is an ordered set of column names of size m and \mathcal{R} is a bag of m -sized tuples with values from $V \cup \mathbb{R}$ denoting the rows. The size of D is equal to the size of \mathcal{R} .

Intuitively, an RDFFrame is a subset of information extracted from one or more knowledge graphs. The rows of an RDFFrame should contain values that are either (a) URIs or literals in a knowledge graph, or (b) aggregated values on data extracted from a graph. Due to the bag semantics, an RDFFrame may contain duplicate rows, which is good in machine learning because it preserves the data distribution and is compatible with the bag semantics of SPARQL.

3.2 API Operators

RDFFrames provides the user with two types of operators: (a) exploration and navigational operators, which operate on a knowledge graph, and (b) relational operators, which operate on an RDFFrame (or two in case of joins). The full list of operators, and also other RDFFrames functions (e.g., for client-server communication), can be found with the source code.¹⁷

The RDFFrames exploration operators are needed to deal with one of the challenges of real-world knowledge graphs: knowledge graphs in RDF are typically multi-topic, heterogeneous, incomplete, and sparse, and the data distributions can be highly skewed. Identifying a relatively small, topic-focused dataset from such a knowledge graph to extract into an RDFFrame is not a simple task, since it requires knowing the structure and schema of the dataset. RDFFrames provides data exploration operators to help with this task. For example, RDFFrames includes operators to identify the RDF classes representing entity types in a knowledge graph, and to compute the data distributions of these classes.

Guided by the initial exploration of the graph, the user can gradually build an RDFFrame representing the information to be extracted. The first step is always a call to the *seed* operator (described below) that initializes the RDFFrame with columns from the knowledge graph. The rest of the RDFFrame is built through a sequence of calls to the RDFFrames navigational and relational operators. Each of these operators outputs an

RDFFrame. The inputs to an operator can be a knowledge graph, one or more RDFFrames, and/or other information such as predicates or column names.

The RDFFrames navigational operators are used to extract information from a knowledge graph into a tabular form using a navigational, procedural interface. RDFFrames also provides relational operators that apply operations on an RDFFrame such as filtering, grouping, aggregation, filtering based on aggregate values, sorting, and join. These operators do not access the knowledge graph, and one could argue that they are not necessary in RDFFrames since they are already provided by machine learning tools that work on dataframes such as pandas. However, we opt to provide these operators in RDFFrames so that they can be pushed into the RDF engine, which results in substantial performance gains as we will see in Section 6.

In the following, we describe the syntax and semantics of the main operators of both types. Without loss of generality, let $G = (V, E)$ be the input knowledge graph and $D = (\mathcal{C}, \mathcal{R})$ be the input RDFFrame of size n . Let $D' = (\mathcal{C}', \mathcal{R}')$ be the output RDFFrame. In addition, let \bowtie , \bowtie_L , \bowtie_R , \bowtie_{full} , σ , π , ρ , and γ be the inner join, left outer join, right outer join, full outer join, selection, projection, renaming, and grouping-with-aggregation relational operators, respectively, defined using bag semantics as in typical relational databases [15].

Exploration and Navigational Operators. These operators traverse a knowledge graph to extract information from it to either construct a new RDFFrame or expand an existing one. They bridge the gap between the RDF data model and the tabular format by allowing the user to extract tabular data through graph navigation. They take as input either a knowledge graph G , or a knowledge graph G and an RDFFrame D , and output an RDFFrame D' .

- $G.seed(col_1, col_2, col_3)$ where col_1, col_2, col_3 are in $N \cup V$: This operator is the starting point for constructing any RDFFrame. Let $t = (col_1, col_2, col_3)$ be a SPARQL triple pattern, then this operator creates an initial RDFFrame by converting the evaluation of the triple pattern t on graph G to an RDFFrame. The returned RDFFrame has a column for every variable in the pattern t . Formally, let D_t be the RDFFrame equivalent to the evaluation of the triple pattern t on graph G . We formally define this notion of equivalence in Section 5. The returned RDFFrame is defined as $D' = \pi_{N \cap \{col_1, col_2, col_3\}}(D_t)$. As an example, the *seed* operator can be used to retrieve all instances of class type T in graph G by calling $G.seed(instance, rdf:type, T)$. For convenience, RDFFrames provides implementations for the most

¹⁷ <https://github.com/qcri/rdfframes>

common variants of this operator. For example, the `feature_domain_range` operator in Listing 1 initializes the RDFFrame with all pairs of entities in DBpedia connected by the predicate `dbpp:starring`, which are movies and the actors starring in them.

- $(G, D).expand(col, pred, new_col, dir, is_opt)$, where $col \in \mathcal{C}$, $pred \in V$, $new_col \in \mathcal{N}$, $dir \in \{in, out\}$, and $is_opt \in \{true, false\}$: This is the main navigational operator in RDFFrames. It expands an RDFFrame by navigating from col following the edge $pred$ to new_col in direction dir . Depending on the direction of navigation, either the starting column for navigation col is the subject of the triple and the ending column new_col is the object, or vice versa. is_opt determines whether null values are allowed. If is_opt is false, `expand` filters out the rows in D that have a null value in new_col . Formally, if t is a SPARQL pattern representing the navigation step, then $t = (col, pred, new_col)$ if direction is *out* or $t = (new_col, pred, col)$ if direction is *in*. Let D_t be the RDFFrame corresponding to the evaluation of the triple pattern t on graph G . D_t will contain one column new_col and the rows are the objects of t if the direction is *in* or the subjects if the direction is *out*. Then $D' = D \bowtie D_t$ if is_opt is false or $D' = D \bowtie D_t$ if is_opt is true. For example, in Listing 1, `expand` is used twice, once to add the `country` attribute of the actor to the RDFFrame and once to find the movies and (if available) Academy Awards for prolific American actors.

Relational Operators. These operators are used to clean and further process RDFFrames. Their semantics is the same as in relational databases. They take as input one or two RDFFrames and output an RDFFrame.

- $D.filter(conds = [cond_1 \wedge cond_2 \wedge \dots \wedge cond_k])$, where $conds$ is a list of expressions of the form $(col \{<, >, =, \dots\} val)$ or one of the pre-defined boolean functions found in SPARQL like `isURI(col)` or `isLiteral(col)`: This operator filters out rows from an RDFFrame that do not conform to $conds$. Formally, let $\varphi = [cond_1 \wedge cond_2 \wedge \dots \wedge cond_k]$ be a propositional formula where $cond_i$ is an expression. Then $D' = \sigma_\varphi(D)$. In Listing 1, `filter` is used two times, once to restrict the extracted data to American actors and once to restrict the results of a group by in order to identify prolific actors (defined as having 50 or more movies). The latter `filter` operator is applied after `group_by` and the aggregation function `count`, which corresponds to a very different SPARQL pattern compared to the first usage. However, this is handled internally by RDFFrames and is transparent to the user.
- $D.select_cols(cols)$, where $cols \subseteq \mathcal{C}$: Similar to relational projection, it keeps only the columns $cols$ and removes the rest. Formally, $D' = \pi_{cols}(D)$.
- $D.join(D_2, col, col_2, jtype, new_col)$, where $D_2 = (\mathcal{C}_2, \mathcal{R}_2)$ is another RDFFrame, $col \in \mathcal{C}$, $col_2 \in \mathcal{C}_2$, and $jtype \in \{\bowtie, \bowtie, \bowtie, \bowtie\}$: This operator joins two RDFFrame tables on their columns col and col_2 using the join type $jtype$. new_col is the desired name of the new joined column. Formally, $D' = \rho_{new_col/col}(D) \ jtype \ \rho_{new_col/col_2}(D_2)$.
- $D.group_by(group_cols).aggregation(fn, col, new_col)$, where $group_cols \subseteq \mathcal{C}$, $fn \in \{max, min, average, sum, count, sample\}$, $col \in \mathcal{C}$ and $new_col \in \mathcal{N}$: This operator groups the rows of D according to their values in one or more columns $group_cols$. As in the relational grouping and aggregation operation, it partitions the rows of an RDFFrame into groups and then applies the aggregation function on the values of column col within each group. It returns a new RDFFrame which contains the grouping columns and the result of the aggregation on each group, i.e., $\mathcal{C}' = group_cols \cup \{new_col\}$. The combinations of values of the grouping columns in D' are unique. Formally, $D' = \gamma_{group_cols, fn(col) \mapsto new_col}(D)$. Note that query generation has special handling for RDFFrames output by the `group_by` operator (termed **grouped RDFFrames**). This special handling is internal to RDFFrames and transparent to the user. In Listing 1, `group_by` is used with the `count` function to find the number of movies in which each actor appears.
- $D.aggregate(fn, col, new_col)$, where $col \in \mathcal{C}$ and $fn \in \{max, min, average, sum, count, distinct_count\}$: This operator aggregates values of the column col and returns an RDFFrame that has one column and one row containing the aggregated value. It has the same formal semantics as the $D.group_by().aggregation()$ operator except that $group_cols = \emptyset$, so the whole RDFFrame is assumed to be one group. No further processing can be done on the RDFFrame after this operator.
- $D.sort(cols_order)$, where $cols_order$ is a set of pairs $(col, order)$ with $col \in \mathcal{C}$ and $order \in \{asc, desc\}$: This operator sorts the rows of the RDFFrame according to their values in the given columns and their sorting order and returns a sorted RDFFrame.
- $D.head(k, i)$, where $k \leq n$: Returns the first k rows of the RDFFrame starting from row i (by default $i = 0$). No further processing can be done on the RDFFrame after this operator.

4 Query Generation

One of the key innovations in RDFFrames is the query generation process. Query generation produces a SPARQL query from an RDFFrame representing a sequence of calls to RDFFrames operators. The guidelines we use in query generation to guarantee efficient processing are as follows:

- Include all of the computation required for generating an RDFFrame in the SPARQL query sent to the RDF engine. Pushing computation into the engine enables RDFFrames to take advantage of the benefits of a database system such as query optimization, bulk data processing, and near-data computing.
- Generate one SPARQL query for each RDFFrame, never more. RDFFrames combines all graph patterns and operations described by an RDFFrame into a single SPARQL query. This minimizes the number of interactions with the RDF engine and enables the query optimizer to explore all optimization opportunities since it can see all operations.
- Ensure that the generated query is as simple as possible. The query generation algorithm generates graph patterns that minimize the use of nested subqueries and union SPARQL patterns, since these are known to be expensive. Note that, in principle, we are doing part of the job of the RDF engine’s query optimizer. A powerful-enough optimizer would be able to simplify and unnest queries whenever possible. However, the reality is that SPARQL is a complex language on which query optimizers do not always do a good job. As such, any steps to help the query optimizer are of great use. We show the performance benefit of this approach in Section 6.
- Adopt a lazy execution model, generating and processing a query only when required by the user.
- Ensure that the generated SPARQL query is correct, that is, ensure the query is semantically equivalent to the RDFFrame. We prove this in Section 5.

Query generation in RDFFrames is a two-step process. First, the sequence of operators describing the RDFFrame is converted to an intermediate representation that we call the *query model*. Second, the query model is traversed to generate the SPARQL query.

4.1 Query Model

Our query model is inspired by the Query Graph Model [36], and it encapsulates all components required to construct a SPARQL query. Query models can be nested in cases where nested subqueries are required. Using the query model as an intermediate representation between an RDFFrame and the corresponding SPARQL query allows for (i) flexible implementation by separating the operator manipulation logic

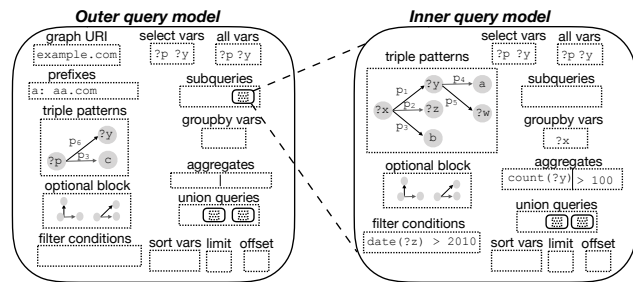


Fig. 2: Example of an RDFFrames nested query model.

from the query generation logic, and (ii) simpler optimization. Without a query model, a naive implementation of RDFFrames would translate each operator to a SPARQL pattern and encapsulate it in a subquery, with one outer query joining all the subqueries to produce the result. This is analogous to how some software-generated SQL queries are produced. Other implementations are possible such as producing a SPARQL query for each operator and re-parsing it every time it has to be combined with a new pattern, or directly manipulating the parse tree of the query. The query model enables a simpler and more powerful implementation.

An example query model representing a nested SPARQL query is shown in Figure 2. The left part of the figure is the outer query model, which has a reference to the inner query model (right part of the figure). The figure shows the components of a SPARQL query represented in a query model. These are as follows:

- Graph matching patterns including triple patterns, filter conditions, pointers to inner query models for sub-queries, optional blocks, and union patterns. Graph pattern matching is a basic operation in SPARQL. A SPARQL query can be formed by combining triple patterns in various ways using different keywords. The default is that a solution is produced if and only if every triple pattern that appears in a graph pattern is matched to the triples in the RDF graph. The `OPTIONAL` keyword adds triple patterns that extend the solution if they are matched, but do not eliminate the solution if they are not matched. That is, `OPTIONAL` creates left outer join semantics. The `FILTER` keyword adds a condition and restricts the query results to solutions that satisfy this condition.
- Aggregation constructs including: group-by columns, aggregation columns, and filters on aggregations (which result in a `HAVING` clause in the SPARQL query). These patterns are applied to the result RDFFrame generated so far. Unlike graph matching patterns, they are not matched to the RDF graph. Aggregation constructs in inner query models are not propagated to outer query models.

- Query modifiers including limit, offset and sorting columns. These constructs make final modifications to the result of the query. Any further API calls after adding these modifiers will result in a nested query as the current query model is wrapped and added to another query model.
- The graph URIs by the query, the prefixes used, and the variables in the scope of each query.

4.2 Query Model Generation

The query model is generated lazily, when the special `execute` function is called on an RDFFrame. We observe that generating the query model requires capturing the order of calls to RDFFrames operators and the parameters of these calls, but nothing more. Thus, each RDFFrame D created by the user is associated with a FIFO queue of operators. The **Recorder** component of RDFFrames (recall Figure 1) records in this queue the sequence of operator calls made by the user. When `execute` is called, the **Generator** component of RDFFrames creates the query model incrementally by processing the operators in this queue in FIFO order. RDFFrames starts with an empty query model m . For each operator pulled from the queue of D , its corresponding SPARQL component is inserted into m . Each RDFFrames operator edits one or two components of m . *All of the optimizations to generate efficient SPARQL queries are done during query model generation.*

The first operator to be processed is always a *seed* operator for which RDFFrames adds the corresponding triple pattern to the query model m . To process an *expand* operator, it adds the corresponding triple pattern(s) to m . For example, the operator `expand(x , $pred$, y , out, false)` will result in the triple pattern $(?x, pred, ?y)$ being added to the triple patterns of m . Similarly, processing the *filter* operator adds the conditions that are input parameters of this operator to the filter conditions in m . To generate succinct optimized queries, RDFFrames adds all triple and filter patterns to the same query model m , as long as the semantics are preserved. As a special case, when *filter* is called on an aggregated column, the **Generator** adds the filtering condition to the *having* component of m .

One of the main challenges in designing RDFFrames was identifying cases where a nested SPARQL query is necessary. We were able to limit to three cases where a nested query is needed for correct semantics:

- Case 1: When an *expand* or *filter* operator has to be applied on a grouped RDFFrame. The semantics here can be thought of as creating an RDFFrame that satisfies the expand or filter pattern and then joining it with the grouped RDFFrame. For example, the RDFFrames code in Listing 3 expands the

country column to obtain the *continent* after the `group_by` and `count`. This is semantically equivalent to building an RDFFrame of countries and their continents and then performing an inner join with the grouped RDFFrame.

```
df = graph.entities(':dpo:Actor', 'actor')\
    .expand('actor', [('dbp:birthPlace', 'country')\
                    ])\
    .group_by(['actor'])\
    .count('country', 'country_count')\
    .expand('country', [('dbo:continent', 'continent')\
                    ])
```

Listing 3: RDFFrames code - Expanding a grouped RDFFrame.

- Case 2: When a grouped RDFFrame has to be joined with another RDFFrame (grouped or non-grouped). For example, Listing 4 represents a join between a grouped RDFFrame and another RDFFrame.

```
df1 = graph.entities('dbo:Actor', 'actor')\
    .expand('actor', [('dbp:birthPlace', 'country')\
                    ])\
    .group_by(['actor']).count('country', 'country_count')
df2 = graph.feature_domain_range('dbp:starring', 'movie', 'actor').join(df1, 'actor', InnerJoin)
```

Listing 4: RDFFrames code - Joining a grouped RDFFrame with another RDFFrame.

- Case 3: When two datasets are joined by a full outer join. For example, the RDFFrames code in Listing 5 is a full outer join between two datasets.

```
df1 = graph.entities(':dpo:Actor', 'actor')\
    .expand('actor', [('dbp:birthPlace', 'country')\
                    ])\
df2 = graph.feature_domain_range('dbp:starring', 'movie', 'actor').join(df1, 'actor', OuterJoin)
```

Listing 5: RDFFrames code - Full outer join.

There is no explicit full outer join between patterns in SPARQL, only left outer join using the OPTIONAL pattern. Therefore, we define full outer join using the UNION and OPTIONAL patterns as the union of the left outer join and the right outer join of D_1 and D_2 . A nesting query is required to wrap the query model for each RDFFrame inside the final query model.

In the first case, when an *expand* operation is called on a grouped RDFFrame, RDFFrames has to wrap the grouped RDFFrame in a nested subquery to ensure the evaluation of the grouping and aggregation operations before the expansion. RDFFrames uses the following steps to generate the subquery: (i) create an empty query model m' , (ii) transform the query model built so far m into a subquery of m' , and (iii) add the new triple pattern from the *expand* operator to the triple patterns of m' . In this case, m' is the outer query model after the *expand* operator and the grouped RDFFrame is represented by the inner query model m . Similarly, when *filter* is applied on a grouping column in a grouped

RDFFrame, RDFFrames creates a nested query model by transforming m into a subquery. This is necessary since the filter operation was called after the aggregation and, thus, has to be done after the aggregation to maintain the correctness of the aggregated values.

The second case in which a nested subquery is required is when joining a grouped RDFFrame with another RDFFrame. In the following, we describe in full the different cases of processing the *join* operator, including the cases when subqueries are required.

To process the binary *join* operator, RDFFrames needs to join two query models of two different RDFFrames D_1 and D_2 . If the join type is full outer join, a complex query that is equivalent to the full outer join is constructed using the SPARQL OPTIONAL (\bowtie) and UNION (\cup) patterns. Formally, $D_1 \bowtie D_2 = (D_1 \bowtie D_2) \cup \rho_{reorder}(D_2 \bowtie D_1)$.

To process a full outer join, two new query models are constructed: The first query model m_1' contains the left outer join of the query models m_1 and m_2 , which represent D_1 and D_2 , respectively. The second query model m_2' contains the right outer join of the query models m_1 and m_2 , which is equivalent to the left outer join of m_2 and m_1 . The columns of m_2' are reordered to make them union compatible with m_1' . Nested queries are necessary to wrap the two query models m_1 and m_2 inside m_1' and m_2' . One final outer query model unions the two new query models m_1' and m_2' .

For other join types, we distinguish three cases:

- D_1 and D_2 are not grouped: RDFFrames merges the two query models into one by combining their graph patterns (e.g., triple patterns and filter conditions). If the join type is left outer join, the patterns of D_2 are added inside a single OPTIONAL block of D_1 . Conversely, for a right outer join the D_1 patterns are added as OPTIONAL in D_2 . No nested query is generated here.
- D_1 is grouped and D_2 is not: RDFFrames merges the two query models via nesting. The query model of D_1 is the inner query model, while D_2 is set as the outer query model. If the join type is left outer join, D_2 patterns are wrapped inside a single OPTIONAL block of D_1 , and if the join type is right outer join, the subquery model generated for D_1 is wrapped in an OPTIONAL block in D_2 . This is an example of the second case in which nested queries are necessary. The case when D_2 is grouped and D_1 is not is analogous to this case.
- Both D_1 and D_2 are grouped: RDFFrames creates one query model containing two nested query models, one for each RDFFrame, another example of the second case where nested queries are necessary.

If D_1 and D_2 are constructed from different graphs, the original graph URIs are used in the inner query to map each pattern to the graph it is supposed to match.

To process other operators such as *select_cols* and *group_by*, RDFFrames fills the corresponding component in the query model. The *head* operator maps to the *limit* and *offset* components of the query model m . To finalize the join processing, RDFFrames unions the selection variables of the two query models, and takes the minimum of the offsets and maximum of the limits (in case both query models have offset and limit).

4.3 Translating to SPARQL

The query model is designed to make the translation to SPARQL as direct and simple as possible. RDFFrames traverses a query model and translates each component of the model directly to the corresponding SPARQL construct, following the syntax and style guidelines of SPARQL.

For example, each prefix is translated to `PREFIX name_space:name_space_uri`, graph URIs are added to the `FROM` clause, and each triple and filter pattern is added to the `WHERE` clause. The inner query models are translated recursively to SPARQL queries and added to the outer query using the subquery syntax defined by SPARQL. When the query accesses more than one graph and different subsets of graph patterns are matched to different graphs, the `GRAPH` construct is used to wrap each subset of graph patterns with the matching graph URI.

The generated SPARQL query is sent to the RDF engine or SPARQL endpoint using the SPARQL protocol¹⁸ over HTTP. We choose communication over HTTP since it is the most general mechanism to communicate with RDF engines and the only mechanism to communicate with SPARQL endpoints. One issue we need to address is paginating the results of a query, that is, retrieving them in chunks. There are several good reasons to paginate results, for example, avoiding timeouts at SPARQL endpoints and bounding the amount of memory used for result buffering at the client. When using HTTP communication, we cannot rely on RDF engine cursors to do the pagination as they are engine-specific and not supported by the SPARQL protocol over HTTP. The HTTP response returns only the first chunk of the result and the size of the chunk is limited by the SPARQL endpoint configuration. The SPARQL over HTTP client has to ask for the rest of the result chunk by chunk but this functionality is not implemented by many existing clients. Since our goal is generality and flexibility, RDFFrames implements pagination

¹⁸ <https://www.w3.org/TR/sparql11-protocol>

transparently to the user and returns one dataframe with all the query results.

5 Semantic Correctness of Query Generation

In this section, we formally prove that the SPARQL queries generated by RDFFrames return results that are consistent with the semantics of the RDFFrames operators. We start with an overview of RDF and the SPARQL algebra to establish the required notation. We then summarize the semantics of SPARQL, which is necessary for our correctness proof. Finally, we formally describe the query generation algorithm in RDFFrames and prove its correctness.

5.1 SPARQL Algebra

The RDF data model can be defined as follows. Assume there are countably infinite pairwise disjoint sets I , B , and L representing URIs, blank nodes, and literals, respectively. Let $T = (I \cup B \cup L)$ be the set of RDF terms. The basic component of an RDF graph is an RDF triple $(s, p, o) \in (I \cup B) \times I \times T$ where s is the *subject*, o is the *object*, and p is the *predicate*. An *RDF graph* is a finite set of RDF triples. Each triple represents a fact describing a relationship of type *predicate* between the *subject* and the *object* nodes in the graph.

SPARQL is a graph-matching query language that evaluates patterns on graphs and returns a result set. Its algebra consists of two building blocks: *expressions* and *patterns*. Let $X = \{?x_1, ?x_2, \dots, ?x_n\}$ be a set of variables disjoint from the RDF terms T , the SPARQL syntactic blocks are defined over T and X . For a pattern P , $Var(P)$ is the set of all variables in P . Expressions and patterns are defined recursively as follows:

- A triple $t \in (I \cup L \cup X) \times (I \cup X) \times (I \cup L \cup X)$ is a pattern.
- If P_1 and P_2 are patterns, then $P_1 \text{ Join } P_2$, $P_1 \text{ Union } P_2$, and $P_1 \text{ LeftJoin } P_2$ are patterns.
- Let all variables in X and all terms in $I \cup L$ be SPARQL expressions; then $(E1 + E2)$, $(E1 - E2)$, $(E1 \times E2)$, $(E1/E2)$, $(E1 = E2)$, $(E1 < E2)$, $(\neg E1)$, $(E1 \wedge E2)$, and $(E1 \vee E2)$ are expressions. If P is a pattern and E is an expression then $Filter(E, P)$ is a pattern.
- If P is a pattern and X is a set of variables in $Var(P)$, then $Project(X, P)$ and $Distinct(Project(X, P))$ are patterns. These two constructs allow nested queries in SPARQL and by adding them, there is no meaningful distinction between SPARQL patterns and queries.
- If P is a pattern, E is an expression and $?x$ is a variable not in $Var(P)$, then $Extend(?x, E, P)$ is a pattern. This allows assignment of expression values

to new variables and is used for variable renaming in RDFFrames.

- If X is a set of variables, $?z$ is another variable, f is an aggregation function, E is an expression, and P is a pattern, then $GroupAgg(X, ?z, f, E, P)$ is a pattern where X is the set of grouping variables, $?z$ is a fresh variable to store the aggregation result, E is often a variable that we are aggregating on. This pattern captures the grouping and aggregation constructs in SPARQL 1.1. It induces a partitioning of a pattern's solution mappings into equivalence classes based on the values of the grouping variables and finds one aggregate value for each class using one of the aggregation functions in $\{max, min, average, sum, count, sample\}$.

SPARQL defines some modifiers for the result set returned by the evaluation of the patterns. These modifiers include: $Order(X, order)$ where X is the set of variables to sort on and $order$ is *ascending* or *descending*, $Limit(n)$ which returns the first n values of the result set, and $Offset(k)$ which returns the results starting from the k -th value.

5.2 SPARQL Semantics

In this section, we present the semantics defined in [23], which assume bag semantics and integrate all SPARQL 1.1 features such as aggregation and subqueries.

The semantics of SPARQL queries are based on multisets (bags) of mappings. A **mapping** is a partial function μ from X to T where X is a set of variables and T is the set of RDF terms. The domain of a mapping $dom(\mu)$ is the set of variables where μ is defined. μ_1 and μ_2 are compatible mappings, written $(\mu_1 \sim \mu_2)$, if $(\forall ?x \in dom(\mu_1) \cap dom(\mu_2), \mu_1(?x) = \mu_2(?x))$. If $\mu_1 \sim \mu_2$, $\mu_1 \cup \mu_2$ is also a mapping and is obtained by extending μ_1 by μ_2 mappings on all the variables $dom(\mu_2) \setminus dom(\mu_1)$. A SPARQL pattern solution is a multiset $\Omega = (S_\Omega, card_\Omega)$ where S_Ω is the base set of mappings, and the multiplicity function $card_\Omega$ assigns a positive number to each element of S_Ω .

Let $\llbracket E \rrbracket_G$ denote the evaluation of expression E on graph G , $\mu(P)$ the pattern obtained from P by replacing its variables according to μ , and $Var(P)$ the set of all the variables in P . The semantics of patterns over graph G are defined as:

- $\llbracket t \rrbracket_G$: the solution of a triple pattern t is the multiset with $S_t = \text{all } \mu \text{ such that } dom(\mu) = Var(t) \text{ and } \mu(t) \in G$. $card_{\llbracket t \rrbracket_G}(\mu) = 1$ for all such μ .
- $\llbracket P_1 \text{ Join } P_2 \rrbracket_G = \{\{\mu | \mu \in \llbracket P_1 \rrbracket_G, \mu_2 \in \llbracket P_2 \rrbracket_G, \mu = \mu_1 \cup \mu_2\}\}$
- $\llbracket P_1 \text{ LeftJoin } P_2 \rrbracket_G = \{\{\mu | \mu \in \llbracket P_1 \text{ Join } P_2 \rrbracket_G\} \uplus \{\{\mu | \mu \in \llbracket P_1 \rrbracket_G, \forall \mu_2 \in \llbracket P_2 \rrbracket_G, (\mu \approx \mu_2)\}\}\}$
- $\llbracket P_1 \text{ Union } P_2 \rrbracket_G = \llbracket P_1 \rrbracket_G \uplus \llbracket P_2 \rrbracket_G$

- $\llbracket Filter(E, P) \rrbracket_G = \{\{\mu | \mu \in \llbracket P \rrbracket_G, \llbracket E \rrbracket_{\mu, G} = true\}\}$
- $\llbracket Project(X, P) \rrbracket_G = \forall \mu \in \llbracket P \rrbracket_G$, if μ is a restriction to X then it is in the base set of this pattern and its multiplicity is the sum of multiplicities of all corresponding μ .
- $\llbracket Distinct(Q) \rrbracket_G$ = the multiset with the same base set as $\llbracket Q \rrbracket_G$, but with multiplicity 1 for all mappings. The SPARQL patterns $Project(X, P)$ and $Distinct(Project(X, P))$ define a SPARQL query. When used in the middle of a query, they define a nested query.
- $\llbracket Extend(?x, E, P) \rrbracket_G = \{\mu' | \mu \in \llbracket P \rrbracket_G, \mu' = \mu \cup \{?x \rightarrow \llbracket E \rrbracket_{\mu, G}\}, \llbracket E \rrbracket_{\mu, G} \neq Error\} \uplus \{\mu | \mu \in \llbracket P \rrbracket_G, \llbracket E \rrbracket_{\mu, G} = Error\}$ and $Var(Extend(?x, E, P)) = \{?x\} \cap Var(P)$
- Given a graph G , let $v|x$ be the restriction of v to X , then $\llbracket GroupAgg(X, ?z, f, E, P) \rrbracket_G$ is the multiset with the base set: $\{\mu' | \mu' = \mu | X \cup \{?z \rightarrow v_\mu\}, \mu \in \llbracket P \rrbracket_G, v_\mu \neq Error\} \cup \{\mu' | \mu' = \mu | X, \mu \in \llbracket P \rrbracket_G, v_\mu = Error\}$ and multiplicity 1 for each mapping in the base set, where for each mapping $\mu \in \llbracket P \rrbracket_G$, the value of the aggregation function on the group that the mapping belongs to is $v_\mu = f(\{v | \mu' \in \llbracket P \rrbracket_G, \mu' | x = \mu | x, v = \llbracket E \rrbracket_{\mu', G}\})$.

5.3 Semantic Correctness

Having defined the semantics of SPARQL patterns, we now prove the semantic correctness of query generation in RDFFrames as follows. First, we formally define the SPARQL query generation algorithm. That is, we define the SPARQL query or pattern generated by any sequence of RDFFrames operators. We then prove that the solution sets of the generated SPARQL patterns are equivalent to the RDFFrames tables defined by the semantics of the sequence of RDFFrames operators.

5.3.1 Query Generation Algorithm

To formally define the query generation algorithm, we first define the SPARQL pattern each RDFFrames operator generates. We then give a recursive definition of a non-empty RDFFrame and then define a recursive mapping from any sequence of RDFFrames operators constructed by the user to a SPARQL pattern using the patterns generated by each operator. This mapping is based on the query model described in Section 4.

Definition 3 (Non-empty RDFFrame) A non-empty RDFFrame is either generated by the *seed* operator or by applying an RDFFrames operator on one or two non-empty RDFFrames.

Given a non-empty RDFFrame D , let O_D be the sequence of RDFFrames operators that generated it.

Definition 4 (Operators to Patterns) Let $O = [o_1, \dots, o_k]$ be a sequence of RDFFrames operators and P be a SPARQL pattern. Also let $g : (o, P) \rightarrow P$ be the mapping from a single RDFFrames operator o to a SPARQL pattern based on the query generation of RDFFrames described in Section 4, also illustrated in Table 1. Mapping g takes as input an RDFFrames operator o and a SPARQL pattern P corresponding to the operators done so far on an RDFFrame D , applies a SPARQL operator defined by the query model generation algorithm on the input SPARQL pattern P , and returns a new SPARQL pattern. Using g , we define a recursive mapping F on a sequence of RDFFrames operators $O, F : O \rightarrow P$, as:

$$F(O) = \begin{cases} g(o_1, Null), & \text{if } |O| \leq 1. \\ g(o_k, F(O_{[1:k-1]}), F(O_{D_2})), & o_k = join(D_2, \dots). \\ g(o_k, F(O_{[1:k-1]})), & \text{otherwise.} \end{cases} \quad (1)$$

F returns a triple pattern for the seed operator and then builds the rest of the SPARQL query by iterating over the RDFFrames operators according to their order in the sequence O .

5.4 Proof of Correctness

To prove the equivalence between the SPARQL pattern solution returned by F and the RDFFrame generating it, we first define the meaning of equivalence between a relational table with bag semantics and the solution sets of SPARQL queries. First, we define a mapping that converts SPARQL solution sets to relational tables by letting the domains of the mappings be the columns and their ranges be the rows. Next, we define the equivalence between solution sets and relations.

Definition 5 (Solution Sets to Relations) Let $\Omega = (S_\Omega, card_\Omega)$ be a multiset (bag) of mappings returned by the evaluation of a SPARQL pattern and $Var(\Omega) = \{?x; ?x \in dom(\mu), \forall \mu \in S_\Omega\}$ be the set of variables in Ω . Let $L = order(Var(\Omega))$ be the ordered set of elements in $Var(\Omega)$. We define a conversion function $\lambda: \Omega \rightarrow R$, where $R = (C, T)$ is a relation. R is defined such that its ordered set of columns (attributes) are the variables in Ω (i.e., $C = L$), and $T = (S_T, card_T)$ is a multiset of (tuples) of values such that for every μ in S_Ω , there is a tuple $\tau \in S_T$ of length $n = |Var(\Omega)|$ and $\tau_i = \mu(L_i)$. The multiplicity function ($card_T$) is defined such that the multiplicity of τ is equal to the multiplicity of μ in $card_\Omega$.

Definition 6 (Equivalence) A SPARQL pattern solution $\Omega = (S_\Omega, card_\Omega)$ is *equivalent* to a relation $R = (C, T)$, written $(\Omega \equiv R)$, if and only if $R = \lambda(\Omega)$.

Table 1: Mappings of RDFFrames operators on graph G and/or RDFFrame D to SPARQL patterns on G . P is the SPARQL pattern equivalent to the sequence of RDFFrames operators called so far on D (or null if D is new).

RDFFrames Operator O	SPARQL pattern: $g(O, P)$
$seed(col_1, col_2, col_3)$	$Project(Var(t), t)$, where $t = (col_1, col_2, col_3)$
$expand(x, pred, y, out, false)$	$P \bowtie (?x, pred, ?y)$
$expand(x, pred, y, in, false)$	$P \bowtie (?y, pred, ?x)$
$expand(x, pred, y, out, True)$	$P \bowtie (?x, pred, ?y)$
$expand(x, pred, y, in, True)$	$P \bowtie (?y, pred, ?x)$
$join(D_2, col, col_2, \bowtie, new_col)$	$Extend(new_col, col, P) \bowtie Extend(new_col, col_2, P_2)$, $P_2 = F(O_{D_2})$
$join(D_2, col, col_2, \bowtie, new_col)$	$Extend(new_col, col, P) \bowtie Extend(new_col, col_2, P_2)$, $P_2 = F(O_{D_2})$
$join(D_2, col, col_2, \bowtie, new_col)$	$Extend(new_col, col_2, P_2) \bowtie Extend(new_col, col, P)$, $P_2 = F(O_{D_2})$
$join(D_2, col, col_2, \bowtie, new_col)$	$(P_1 \bowtie P_2) \cup (P_2 \bowtie P_1)$, $P_1 = Extend(new_col, col, P)$, $P_2 = Extend(new_col, col_2, F(O_{D_2}))$
$filter(conds = [cond_1 \wedge cond_2 \wedge \dots \wedge cond_k])$	$Filter(conds, P)$
$select_cols(cols)$	$Project(cols, P)$
$groupby(group_cols).$	$Project(group_cols \cup \{new_col\})$,
$aggregation(fn, src_col, new_col)$	$GroupAgg(group_cols, new_col, fn, src_col, P)$
$aggregate(fn, col, new_col)$	$Project(\{new_col\}, GroupAgg(\emptyset, new_col, fn, col, P))$

We are now ready to use this definition to present a lemma that defines the equivalent relational tables for the main SPARQL patterns used in our proof.

Lemma 1 *If P_1 and P_2 are SPARQL patterns, then:*

- $\llbracket (P_1 \text{ Join } P_2) \rrbracket_G \equiv \lambda(\llbracket P_1 \rrbracket_G) \bowtie \lambda(\llbracket P_2 \rrbracket_G)$,
- $\llbracket (P_1 \text{ LeftJoin } P_2) \rrbracket_G \equiv \lambda(\llbracket P_1 \rrbracket_G) \bowtie \lambda(\llbracket P_2 \rrbracket_G)$,
- $\llbracket (P_1 \text{ Union } P_2) \rrbracket_G \equiv \lambda(\llbracket P_1 \rrbracket_G) \bowtie \lambda(\llbracket P_2 \rrbracket_G)$
- $\llbracket (Extend(?x, E, P)) \rrbracket_G \equiv \rho_{?x/E}(\lambda(\llbracket P \rrbracket_G))$
- $\llbracket (Filter(conds, P)) \rrbracket_G \equiv \sigma_{conds}(\lambda(\llbracket P \rrbracket_G))$
- $\llbracket (Project(cols, P)) \rrbracket_G \equiv \Pi_{cols}(\lambda(\llbracket P \rrbracket_G))$
- $\llbracket (GroupAgg(\emptyset, new_col, fn, col, P)) \rrbracket_G \equiv \gamma_{cols, fn(col) \mapsto new_col}(\lambda(\llbracket P \rrbracket_G))$

Proof The proof of this lemma follows from (1) the semantics of SPARQL operators presented in Section 5.2, (2) the well-known semantics of relational operators, (3) Definition 5 which specifies the function λ , and (4) Definition 6 which defines the equivalence between multisets of mappings and relations. For each statement in the lemma, we use the definition of the function λ , the relational operator semantics, and the SPARQL operator semantics to define the relation on the right side. Then we use the definition of SPARQL operators semantic to define the multiset on the left side. Finally, Definition 6 proves the statement.

Finally, we present the main theorem in this section, which guarantees the semantic correctness of the RDFFrames query generation algorithm.

Theorem 1 *Given a graph G , every RDFFrame D that is returned by a sequence of RDFFrames operators $O_D = [o_1, \dots, o_k]$ on G is equivalent to the evaluation of the SPARQL pattern $P = F(O_D)$ on G . In other words, $D \equiv \llbracket F(O_D) \rrbracket_G$.*

Proof We prove that $D \equiv \lambda(\llbracket F(O_D) \rrbracket_G)$ via structural induction on non-empty RDFFrame D . For simplicity, we denote the proposition $D \equiv \llbracket F(O_D) \rrbracket_G$ as $A(D)$.

Base case: Let D be an RDFFrame created by one RDFFrames operator $O_D = [seed(col_1, col_2, col_3)]$. The first operator (and the only one in this case) has to be the *seed* operator since it is the only operator that takes only a knowledge graph as input and returns an RDFFrame. From Table 1:

$$\begin{aligned} F(O_D) &= g(seed(col_1, col_2, col_3), Null) \\ &= Project(Var(t), t) \end{aligned}$$

where $t = (col_1, col_2, col_3)$. By definition of the RDFFrames operators in Section 3, $D = \Pi_{X \cap \{col_1, col_2, col_3\}}(\lambda(\llbracket t \rrbracket_G))$ and by Lemma 1(f), $A(D)$ holds.

Induction hypothesis: Every RDFFrames operator takes as input one or two RDFFrames D_1, D_2 and outputs an RDFFrame D . Without loss of generality, assume that both D_1 and D_2 are non-empty and $A(D_1)$ and $A(D_2)$ hold, i.e., $D_1 \equiv \llbracket F(O_{D_1}) \rrbracket_G$ and $D_2 \equiv \llbracket F(O_{D_2}) \rrbracket_G$.

Induction step: Let $D = D_1.Op(\text{optional } D_2)$, $P_1 = F(O_{D_1})$, and $P_2 = F(O_{D_2})$. We use RDFFrames semantics to define D , the mapping F to define the new pattern P , then Lemma 1 to prove the equivalence between F and D . We present the different cases next.

- If Op is $expand(x, pred, y, out, false)$ then: $D = D_1 \bowtie \lambda(\llbracket t \rrbracket_G)$ according to the definition of the operator in Section 3.2 and Table 1, where t is the triple pattern $(?x, pred, ?y)$. By the induction hypothesis, it holds that $D_1 = \lambda(\llbracket P_1 \rrbracket_G)$. Thus, it holds that $D = \lambda(\llbracket P_1 \rrbracket_G) \bowtie \lambda(\llbracket t \rrbracket_G)$ and by Lemma 1(a), $A(D)$ holds. The same holds when Op is $expand(x, pred, y, in, false)$ except that $t = (?y, pred, ?x)$.

- If Op is $join(D_2, col, col_2, \bowtie, new_col)$ then: $D = \rho_{new_col/col}(D_1) \bowtie \rho_{new_col/col_2}(D_2)$, and by $A(D_1)$, $D_1 = \lambda(\llbracket P_1 \rrbracket_G)$ and $D_2 = \lambda(\llbracket P_2 \rrbracket_G)$. Thus, $D = \rho_{new_col/col}(\lambda(\llbracket P_1 \rrbracket_G)) \bowtie \rho_{new_col/col_2}(\lambda(\llbracket P_2 \rrbracket_G))$ and by Lemma 1(a,c), $A(D)$ holds. The same argument holds for other types of join, using the relevant parts of Lemma 1.
- If Op is $filter(conds = [cond_1 \wedge cond_2 \wedge \dots \wedge cond_k])$ then: $D = \sigma_{conds}(D_1)$, and by $A(D_1)$, $D_1 = \lambda(\llbracket P_1 \rrbracket_G)$. So, $D = \sigma_{conds}(\lambda(\llbracket P_1 \rrbracket_G))$ and by Lemma 1(e), $A(D)$ holds.
- If Op is $groupby(cols).aggregation(f, col, new_col)$ then: $D = \gamma_{cols, f(col) \mapsto new_col}(D_1)$, and by $A(D_1)$, $D_1 = \lambda(\llbracket P_1 \rrbracket_G)$. So, $D = \gamma_{cols, f(col) \mapsto new_col}(\lambda(\llbracket P_1 \rrbracket_G))$ and by Lemma 1(f,g), $A(D)$ holds.

Thus, $A(D)$ holds in all cases.

6 Evaluation

We present an experimental evaluation of RDFSFrames in which our goal is to answer two questions: (1) How effective are the design decisions made in RDFSFrames? and (2) How does RDFSFrames perform compared to alternative baselines?

We use two workloads for this experimental study. The first is made up of three case studies consisting of machine learning tasks on two real-world knowledge graphs. Each task starts with a data preparation step that extracts a pandas dataframe from the knowledge graph. This step is the focus of the case studies. In the next section, we present the RDFSFrames code for each case study and the corresponding SPARQL query generated by RDFSFrames. As in our motivating example, we will see that the SPARQL queries are longer and more complex than the RDFSFrames code, thereby showing that RDFSFrames can indeed simplify access to knowledge graphs. The full Python code for the case studies can be found in Appendix A. The second workload in our experiments is a synthetic workload consisting of 16 queries. These queries are designed to exercise different features of RDFSFrames for the purpose of benchmarking. We describe the two workloads next, followed by the experimental setup and the results.

6.1 Case Studies

6.1.1 Movie Genre Classification

Classification is a basic supervised machine learning task. This case study applies a classification task on movie data extracted from the DBpedia knowledge graph. Many knowledge graphs, including DBpedia, are heterogeneous, with information about diverse topics, so extracting a topic-focused dataframe for classification is challenging.

This task uses RDFSFrames to build a dataframe of movies from DBpedia, along with a set of movie attributes that can be used for movie genre classification. The task bears some similarity to the code in Listing 1. Let us say that the classification dataset that we want includes movies that star American actors (since they are assumed to have a global reach) or prolific actors (defined as those who have starred in 100 or more movies). We want the movies starring these actors, and for each movie, we extract the movie name (i.e., title), actor name, topic, country of production, and genre. Genre is not always available so it is an optional predicate. The full code for this data preparation step is shown in Listing 6, and the SPARQL query generated by RDFSFrames is shown in Listing 7.

The extracted dataframe can be used as a classification dataset by any popular Python machine learning library. The movies that have the genre available in the dataframe can be used as labeled training data to train a classifier. The features for this classifier are the attributes of the movies and the actors, and the classifier is trained to predict the genre of a movie based on these features. The classifier can then be used to predict the genres of all movies that are missing the genre.

Note that the focus of RDFSFrames is the data preparation step of a machine learning pipeline (i.e., creating the dataframe). That is, RDFSFrames addresses the following problem: Most machine learning pipelines require as their starting point an input dataframe, and there is no easy way to get such a dataframe from a knowledge graph while leveraging an RDF engine. Thus, the focus of RDFSFrames is enabling the user to obtain a dataframe from an RDF engine, and not how the machine learning pipeline uses this dataframe. Nevertheless, it is interesting to see this dataframe within an end-to-end machine learning pipeline. Specifically, for the current case study, can the dataframe created by RDFSFrames be used for movie genre classification? We emphasize that the accuracy of the classifier is not our main concern here; our concern is demonstrating RDFSFrames in an end-to-end machine learning pipeline. Issues such as using a complex classifier, studying feature importance, or analyzing the distribution of the retrieved data are beyond the scope of RDFSFrames.

To show RDFSFrames in an end-to-end machine learning pipeline, we built a classifier based on the output of Listing 6 to classify the six most frequent movie genres, specifically, drama, sitcom, science fiction, legal drama, comedy, and fantasy. The classification dataset consisted of 7,635 movies that represent the English movies in these six movie genres. We trained a random forest classifier using the scikit-learn machine learning

library based on movie features such as actor country, movie country, subject, and actor name. This classifier achieved 92.4% accuracy on evaluation data that is 30% of the classification dataset.

We performed a similar experiment on song data from DBpedia. We extracted 27,956 triples of English songs in DBpedia along with their features such as album, writer, title, artist, producer, album title, and studio. We used the same methodology as in the movie genre classification case study to classify songs into genres such as alternative rock, hip hop, indie rock, and pop-punk. The accuracy achieved by a random forest classifier in this case was 70.9%.

```
movies = graph.feature_domain_range('dbpp:starring',
    'movie', 'actor')
movies = movies.expand('actor', [(('dbpp:birthPlace',
    'actor_country'), ('rdfs:label', 'actor_name'))]\
    .expand('movie', [(('rdfs:label', 'movie_name'),
    ('dcterms:subject', 'subject'),
    ('dbpp:country', 'movie_country'),
    ('dbpo:genre', 'genre', Optional))]).cache()
american = movies.filter({'actor_country':\
    ['dbpr:UnitedStates']})
prolific = movies.group_by(['actor'])\
    .count('movie', 'movie_count', unique=True)\
    .filter({'movie_count': ['>=100']})
dataset = american.join(prolific, 'actor', OuterJoin)\
    .join(movies, 'actor', InnerJoin)
```

Listing 6: RDFFrames code - Movie genre classification.

```
SELECT DISTINCT ?actor_name ?movie_name ?
    actor_country ?genre ?subject
FROM <http://dbpedia.org>
WHERE
{ ?movie dbpp:starring ?actor .
  ?movie rdfs:label ?movie_name .
  ?movie dcterms:subject ?subject .
  ?actor dbpp:birthPlace ?actor_country .
  ?actor rdfs:label ?actor_name
OPTIONAL
{ ?movie dbpp:genre ?genre }
{{ SELECT * WHERE
  {{ SELECT * WHERE
    { ?movie dbpp:starring ?actor .
      ?movie rdfs:label ?movie_name .
      ?movie dcterms:subject ?subject .
      ?actor dbpp:birthPlace ?actor_country .
      ?actor rdfs:label ?actor_name
      FILTER regex(str(?actor_country), "USA")
      OPTIONAL
        { ?movie dbpp:genre ?genre }
    }
  }
}}
OPTIONAL
{ SELECT DISTINCT ?actor (COUNT(DISTINCT ?movie
) AS ?movie_count)
  WHERE
  { ?movie dbpp:starring ?actor .
    ?movie rdfs:label ?movie_name .
    ?movie dcterms:subject ?subject .
    ?actor dbpp:birthPlace ?actor_country .
    ?actor rdfs:label ?actor_name
    OPTIONAL
      { ?movie dbpp:genre ?genre }
  }
  GROUP BY ?actor
  HAVING ( COUNT(DISTINCT ?movie) >= 100 )
}
UNION
{ SELECT * WHERE
  {{ SELECT DISTINCT ?actor (COUNT(DISTINCT ?movie
) AS ?movie_count) WHERE
```

```
{ ?movie dbpp:starring ?actor .
  ?movie rdfs:label ?movie_name .
  ?movie dcterms:subject ?subject .
  ?actor dbpp:birthPlace ?actor_country .
  ?actor rdfs:label ?actor_name
OPTIONAL
{ ?movie dbpp:genre ?genre }
}
GROUP BY ?actor
HAVING ( COUNT(DISTINCT ?movie) >= 100 )
}
OPTIONAL
{ SELECT * WHERE
  { ?movie dbpp:starring ?actor .
    ?movie rdfs:label ?movie_name .
    ?movie dcterms:subject ?subject .
    ?actor dbpp:birthPlace ?actor_country .
    ?actor rdfs:label ?actor_name
    FILTER regex(str(?actor_country), "USA")
    OPTIONAL
      { ?movie dbpp:genre ?genre }
  }
}
}
```

Listing 7: SPARQL query generated by RDFFrames for the code shown in Listing 6.

6.1.2 Topic Modeling

Topic modeling is a statistical technique commonly used to identify hidden contextual topics in the text. In this case study, we use topic modeling to identify the active topics of research in the database community. We define these as the topics of recent papers published by authors who have published many SIGMOD and VLDB papers. This is clearly an artificial definition, but it enables us to study the capabilities and performance of RDFFrames. As stated earlier, we are focused on data preparation not the details of the machine learning task.

```
papers = graph.entities('swrc:InProceedings', 'paper')
papers = papers.expand('paper', [(('dc:creator', \
    'author'), ('dcterms:issued', 'date'), \
    ('swrc:series', 'conference'), \
    ('dc:title', 'title'))]).cache()
authors = papers.filter({'date': ['>=2000'],
    'conference': ['In(dblp:vldb, dblp:sigmod)']})
    .group_by(['author']).count('paper', 'n_papers')\
    .filter({'n_papers': '>=20', 'date': ['>=2010']})\
titles = papers.join(authors, 'author', InnerJoin)\
    .select_cols(['title'])
```

Listing 8: RDFFrames code - Topic modeling.

```
SELECT ?title
FROM <http://dblp.13s.de>
WHERE
{ ?paper dc:title ?title ;
  rdf:type swrc:InProceedings ;
  dcterms:issued ?date ;
  dc:creator ?author
  FILTER ( year(xsd:dateTime(?date)) >= 2005 )
  { SELECT ?author
    WHERE
    { ?paper rdf:type swrc:InProceedings
      ;
      swrc:series ?conference ;
      dc:creator ?author ;
      dcterms:issued ?date
    }
    FILTER ( ( year(xsd:dateTime(?date)) >=
      2005 )
      && ( ?conference IN (dblp:vldb,
        dblp:sigmod) ) )
  }
}
```



```

GROUP BY ?author
HAVING ( COUNT(?paper) >= 20 )
}
}

```

Listing 9: SPARQL query generated by RDFFrames for the code shown in Listing 8.

The dataframe required for this task is extracted from the DBLP knowledge graph represented in RDF through the sequence of RDFFrames operators shown in Listing 8. First, we identify the authors who have published 20 or more papers in SIGMOD and VLDB since the year 2000, which requires using the RDFFrames grouping, aggregation, and filtering capabilities. For the purpose of this case study, these are considered the thought leaders of the field of databases. Next, we find the titles of all papers published by these authors since 2010. The SPARQL query generated by RDFFrames is shown in Listing 9.

We then run topic modeling on the titles to identify the topics of the papers, which we consider to be the active topics of database research. We use off-the-shelf components from the rich ecosystem of pandas libraries to implement topic modeling (see Appendix A). Specifically, we use NLP libraries for stop-word removal and scikit-learn for topic modeling using SVD. This shows the benefit of using RDFFrames to get data into a pandas dataframe with a few lines of code, since one can then utilize components from the PyData ecosystem.

6.1.3 Knowledge Graph Embedding

Knowledge graph embeddings are widely used relational learning models, and they are state of the art on benchmark datasets for link prediction and fact classification [43, 44]. The input to these models is a dataframe of triples, i.e., a table of three columns: $[subject, predicate, object]$ where the *object* is a URI representing an entity (i.e., not a literal). Currently, knowledge graph embeddings are typically evaluated only on small pre-processed subsets of knowledge graphs like FB15K [7] and WN18 [7] rather than the full knowledge graphs, and thus, the validity of their performance results has been questioned recently in multiple papers [11, 37]. Filtering the knowledge graph to contain only entity-to-entity triples and loading the result in a dataframe is a necessary first step in constructing knowledge graph embedding models on full knowledge graphs. RDFFrames can perform this first step using one line of code as shown in Listing 10 (generated SPARQL in Listing 11). With this line of code, the filtering can be performed efficiently in an RDF engine, and RDFFrames handles issues related to communication with the engine and integrating with PyData. These issues become important, especially when deal-

ing with large knowledge graphs where the resulting dataframe has millions of rows.

```

graph.feature_domain_range(s, p, o)\
    .filter({o: ['isURI']})

```

Listing 10: RDFFrames code - Knowledge graph embedding.

```

SELECT *
FROM <http://dblp.13s.de/>
WHERE {
    ?sub ?pred ?obj .
    FILTER ( isIRI(?obj) )
}

```

Listing 11: SPARQL query corresponding to RDFFrames code shown in Listing 10.

6.2 Synthetic Workload

While the case studies in the previous section show RDFFrames in real applications, it is still desirable to have a more comprehensive evaluation of the framework. To this end, we created a synthetic workload consisting of 16 queries written in RDFFrames that exercise different capabilities of the framework. All the queries are on the DBpedia knowledge graph, and two queries join DBpedia with the YAGO knowledge graph. One query joins the three knowledge graphs DBpedia, YAGO, and DBLP. Four of the queries use only expand and filter (up to 10 expands, including some with optional predicates). Four of the queries use grouping with expand (including one with expand after the grouping). Eight of the queries use joins, including complex queries that exercise features such as outer join, multiple joins, joins between different graphs, and joins on grouped datasets. A description of the queries and the RDFFrames features and SPARQL capabilities that they exercise can be found in Appendix B.

6.3 Experimental Setup

6.3.1 Dataset Details

The three knowledge graphs used in the evaluation have different sizes and statistical features. The first is the English version of the DBpedia knowledge graph. We extracted the December 2020 core collection from DBpedia Databus.¹⁹ The collection contains 6 billion triples. The second is the DBLP computer science bibliography dataset (2017 version) containing 88 million triples.²⁰ The third (used in three queries in the synthetic workload) is YAGO version 3.1, containing 1.6 billion triples. DBLP is relatively small, structured and dense, while DBpedia and YAGO are heterogeneous and sparse.

¹⁹ <https://databus.dbpedia.org/dbpedia/collections/latest-core>

²⁰ <http://www.rdfhdt.org/datasets>

6.3.2 Hardware and Software Configuration

We use an Ubuntu server with 128GB of memory to run a Virtuoso OpenLink Server (version 7.2.6-rc1.3230-pthreads as of Jan 9 2019) with its default configuration. We load the DBpedia, DBLP, and YAGO knowledge graphs to the Virtuoso server. RDFFrames connects to the server to process SPARQL queries over HTTP using `SPARQLWrapper`,²¹ a Python library that provides a wrapper for SPARQL endpoints. Recall that the decision to communicate with the server over HTTP rather than the cursor mechanism of Virtuoso was made to ensure maximum generality and flexibility. When sending SPARQL queries directly to the server, we use the `curl` tool. The client always runs on a separate core of the same machine as the Virtuoso server so we do not incur communication overhead. In all experiments, we report the average running time of three runs.

6.3.3 Alternatives Compared

Our goal is to evaluate the design decisions of RDFFrames and to compare it against alternative baselines. To evaluate the design decisions of RDFFrames, we ask two questions: (1) How important is it to generate optimized SPARQL queries rather than using a simple query generation approach? and (2) How important is it to push the processing of relational operators into the RDF engine? Both of these design choices are clearly beneficial and the goal is to quantify the benefit.

To answer the first question, we compare RDFFrames against an alternative that uses a naive query generation strategy. Specifically, for each API call to RDFFrames, we generate a subquery that contains the pattern corresponding to that API call and we finally join all the subqueries in one level of nesting with one outer query. For example, each call to an `expand` creates a new subquery containing one triple pattern described by the `expand` operator. We refer to this alternative as **Naive Query Generation**. The naive queries for the first two case studies are shown in Appendices C and D. The SPARQL query for the third case study is simple enough that Listing 11 is also the naive query.

To answer the second question, we compare to an alternative that uses RDFFrames (with optimized query generation) only for graph navigation using the `seed` and `expand` operators, and performs any relational-style processing in pandas. We refer to this alternative as **Navigation + pandas**.

If we do not use RDFFrames, we can envision three alternatives for pre-processing the data and loading it into a dataframe, and we compare against all three:

- Do not use an RDF engine at all, but rather write an ad-hoc script that runs on the knowledge graph

stored in some RDF serialization format. To implement this solution we write scripts using the `rdflib` library²² to load the RDF dataset into pandas, and use pandas operators for any additional processing. The `rdflib` library can process any RDF serialization format, and in our case the data was stored in the N-Triples format. We refer to this alternative as **rdflib + pandas**.

- Use an RDF engine, and use a simple SPARQL query to load the RDF dataset into a dataframe. Use pandas for additional processing. This is a variant of the first alternative but it uses SPARQL instead of `rdflib`. The advantage is that the required SPARQL is very simple, but still benefits from the processing capabilities of the RDF engine. We refer to this alternative as **SPARQL + pandas**.
- Use a SPARQL query written by an expert (in this case, the authors of the paper) to do all the pre-processing inside the RDF engine and output the result to a dataframe. This alternative takes full advantage of the capabilities of the RDF engine, but suffers from the “impedance mismatch” described in the introduction: SPARQL uses a different programming style compared to machine learning tools and requires expertise to write, and additional code is required to export the data into a dataframe. We refer to this alternative as **Expert SPARQL**.

We verify that the results of all alternatives are identical. Note that RDFFrames, **Naive Query Generation**, and **Expert SPARQL** generate semantically equivalent SPARQL queries. The query optimizer of an RDF engine should be able to produce query execution plans for all three queries that are identical or at least have similar execution cost. We will see that Virtuoso, being an industry-strength RDF engine, does indeed deliver the same performance for all three queries in many cases. However, we will also see that there are cases where this is not true, which is expected due to the complexity of optimizing SPARQL queries.

6.4 Results on Case Studies

6.4.1 Evaluating the design decisions of RDFFrames

Figure 3 shows the running time of **Naive Query Generation**, **Navigation + pandas**, and RDFFrames on the three case studies.

Movie Genre Classification. This task requires heavy processing on the DBpedia dataset and returns a dataframe of 19,633 movies. The results are presented in Figure 3(a). The running time of RDFFrames is 687.96 seconds. Of this time, less than 5 milliseconds is spent on preparing the SPARQL query (i.e., recording the

²¹ <https://rdflib.github.io/sparqlwrapper>

²² <https://github.com/RDFLib/rdflib>

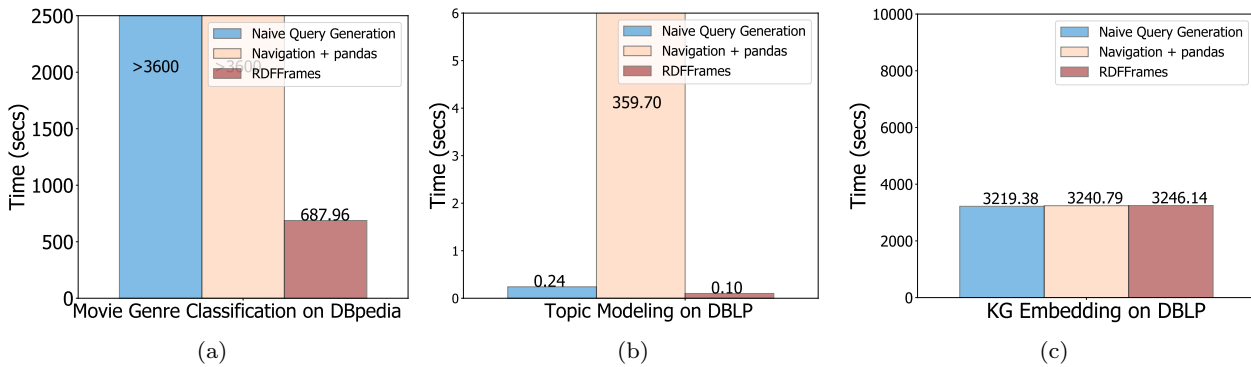


Fig. 3: Evaluating the design of RDFFrames.

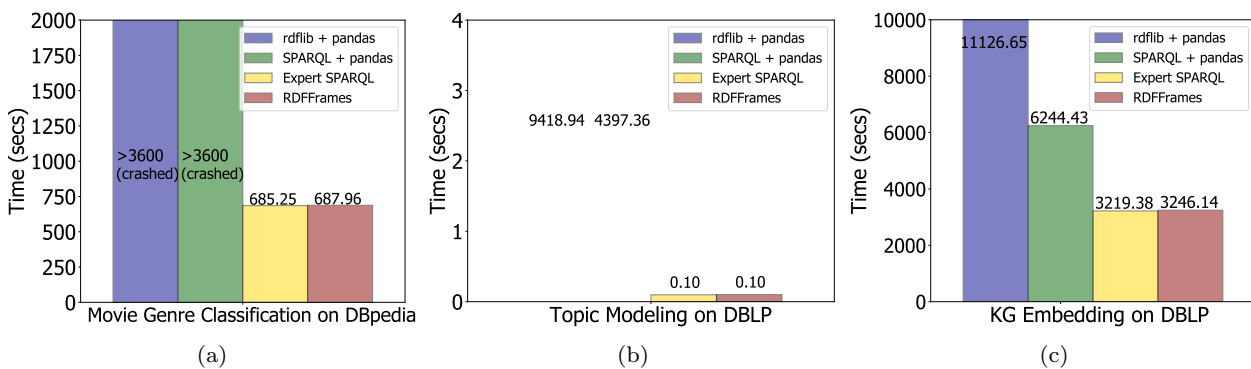


Fig. 4: Comparing RDFFrames to alternative baselines.

RDFFrames operations, generating the query model, and producing the query). The remaining time is spent on issuing the query to the engine and retrieving the results. This is typical in all our experiments: RDFFrames needs a few milliseconds to generate the SPARQL query and the remaining time is spent on query processing. The query produced by naive query generation did not finish in one hour and we terminated it after this time, which demonstrates the need for RDFFrames to generate optimized SPARQL and not rely exclusively on the query optimizer. The `Navigation + pandas` alternative also timed out after one hour, which demonstrates the need for pushing computation into the engine.

Topic Modeling. This task requires heavy processing on the DBLP dataset and returns a dataframe of 4,209 titles. The results are depicted in Figure 3(b). Naive query generation did well here, with the query optimizer generating a good plan for the query. Nonetheless, naive query generation is 2x slower than RDFFrames. This further demonstrates the need for generating optimized SPARQL. The `Navigation + pandas` alternative here was particularly bad, reinforcing the need to push computation into the engine. **Knowledge Graph Embedding.** This task keeps only triples where the object is an entity (i.e., not a literal). It does not require heavy

processing but requires handling the scalability issues of returning a huge final dataframe with all triples of interest. The results on DBLP are shown in Figure 3(c). All the alternatives have similar performance for this task, since the required SPARQL query is simple and processed well by Virtuoso, and since there is no processing required in pandas.

6.4.2 Comparing RDFFrames With Alternative Baselines

Figure 4 compares the running time of RDFFrames on the three case studies to the three alternative baselines: `rdflib + pandas`, `SPARQL + pandas`, and `Expert SPARQL`.

Movie Genre Classification. Both the `rdflib + pandas` and `SPARQL + pandas` baselines crashed after more than one hour due to scalability issues, showing that they are not viable alternatives. On the other hand, RDFFrames and `Expert SPARQL` have similar performance. This shows that RDFFrames does not add overhead and is able to match the performance of an expert-written SPARQL query, which is the best case for an automatic query generator. Thus, the flexibility and usability of RDFFrames does not come at the cost of reduced performance.

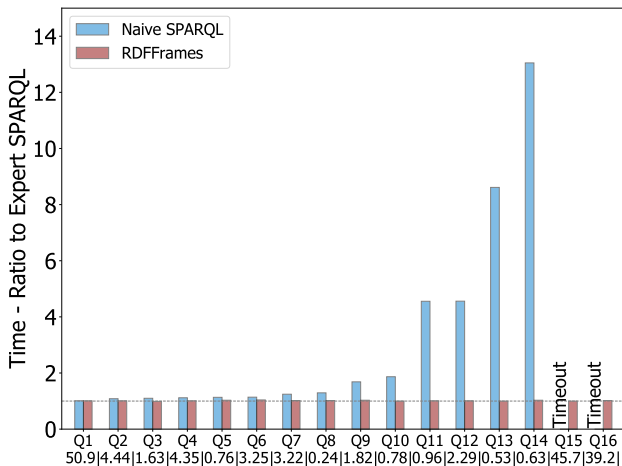


Fig. 5: Results on the synthetic workload.

Topic Modeling. The baselines that perform computation in pandas did not crash as before, but are orders of magnitude slower than RDFFrames and Expert SPARQL. In this case as well, the running time of RDFFrames matches the expert-written SPARQL query.

Knowledge Graph Embedding. In this experiment, `rdflib + pandas` is 3x slower than RDFFrames and `SPARQL + pandas` is 2x slower than RDFFrames, while RDFFrames has the same performance as Expert SPARQL. These results reinforce the conclusions drawn earlier.

6.5 Results on Synthetic Workload

In this experiment, we use the synthetic workload of 16 queries to do a more comprehensive evaluation of RDFFrames. The previous section showed that `Navigation + pandas`, `rdflib + pandas`, and `SPARQL + pandas` are not competitive with RDFFrames. Thus, we exclude them from this experiment. Instead, we focus on the quality of the queries generated by RDFFrames and whether this broad set of queries shows that naive query generation would work well. Figure 5 compares naive query generation and RDFFrames to expert-written SPARQL. The y-axis shows the ratio between the running time of naive query generation and expert-written SPARQL, and between the running time of RDFFrames and expert-written SPARQL. Thus, expert-written SPARQL is considered the gold standard and the figure shows how well the two query generation alternatives match this standard. To improve the comparison, the absolute running time of expert-written SPARQL in seconds is shown under each query on the x-axis. The queries are sorted in ascending order by

the ratio of naive query generation to expert-written SPARQL (the blue bars in the figure). The dashed horizontal line represents a ratio of 1.

The ratios for RDFFrames range between 0.99 and 1.04, which shows that RDFFrames is good at generating queries that match the performance of expert-written queries. On the other hand, the ratios for naive query generation vary widely. The first six queries have ratios between 1.01 and 1.14. For these queries, the Virtuoso optimizer does a good job of generating a close-to-optimal plan for the naive query. The next six queries have ratios between 1.24 and 4.56. Here, we are seeing the weakness of naive query generation and the need for the optimizations performed by RDFFrames during query generation. The situation is worse for the last four queries: naive query generation is an order of magnitude slower for two queries and the last two queries time out after one hour. Thus, the results on this more comprehensive workload validate the quality of the queries generated by RDFFrames and the need for its sophisticated query generation algorithm.

6.6 Effect of Operator Complexity

In our final experiment, we study how the complexity of various RDFFrames operators affects performance. Unlike the previous experiment, in which we varied the complexity of large queries as indicated in Appendix B, this experiment studies the issue of complexity at the granularity of an operator. The cost of operators is highly dependent on the RDF engine being used (Virtuoso in our case). Nevertheless, we want to see if there are any patterns in performance.

To study the effect of operator complexity, we create four RDFFrames queries of increasing complexity that operate on movies in the DBpedia knowledge graph. The performance of these queries is shown in Table 2. The first query counts the number of movies. We find movies by finding entities that are the subject of a ‘starring’ predicate. We then use the *expand* RDFFrames operator to get the movie titles and apply the *count* aggregation function on these titles. The second query uses the *select* RDFFrames operator to retrieve all movie titles. The third query uses the *group.by* operator to group movies by genre and counts the number of movies in each genre. The fourth query is a join query that finds actors who are also movie directors. This query creates a dataset of actors and a dataset of directors, and then joins the two datasets through an inner *join* operator on actor/director name.

We ran the four queries with *filter* operators of varying selectivity. In one case, we had no filter (i.e., we ran the query on all movies). In the second case, we had a filter specifying that movie genre has to be one

Table 2: Running time with varying operator complexity.

filter (on genre)	No. of Movies	Running Time (seconds)			
		count	select	group_by	join
Sitcom	5015	0.088	0.114	0.115	0.342
Sitcom, Drama, Comedy	12115	0.574	0.635	0.704	1.838
No filter (all genres)	87811	3.26	3.163	6.286	77.896

of sitcom, drama, or comedy. In the third case, the filter specified that movie genre has to be sitcom (the most selective filter).

Each row in Table 2 represents a filter, and the number of movies retrieved by this filter is presented in the second column. The rows of the table are sorted by this column. The next four columns in each row show the running time of the four queries for this filter. In all cases, the time that RDFSFrames spends to generate the SPARQL query is less than one millisecond, so we do not report it separately.

Looking at each running time column from top to bottom, we see that the bigger the input data, the more time is required. Looking at each row from left to right, we see that the more complex the query, the more time is required. Both of these results are expected. Another observation about Table 2 is that the variation in running times is not excessive. Even the most expensive query, the join query with no filter, which joins a dataset of 31221 actors with a dataset of 1784 directors, runs in a reasonable 77.896 seconds.

Thus, the experiment shows that operator complexity and dataset size do have an effect on performance. The observed effect is in-line with expectations and does not affect the usability of RDFSFrames. The robust performance we observe is partly due to the robustness of Virtuoso and partly due to our process for SPARQL query generation.

7 Conclusion

We presented RDFSFrames, a framework for seamlessly integrating knowledge graphs into machine learning applications. RDFSFrames is based on a number of powerful operators for graph navigation and relational processing that enable users to generate tabular data sets from knowledge graphs using procedural programming idioms that are familiar in machine learning environments such as PyData. RDFSFrames automatically converts these procedural calls to optimized SPARQL queries and manages the execution of these queries on a local RDF engine or a remote SPARQL endpoint, shielding the user from all details of SPARQL query execution. We provide a Python implementation of RDF-

Frames that is tightly integrated with the pandas library and experimentally demonstrate its efficiency.

Directions for future work include conducting a comprehensive user study to identify and resolve any usability-related issues that could be faced by RDF-Frames users. A big problem in RDF is that users need to know the knowledge graph vocabulary and structure in order to effectively query it. To address this problem, one direction for future work is expanding the exploration operators of RDFSFrames to include keyword search. Testing and evaluating RDFSFrames on multiple RDF engines is another possible future direction.

References

1. Abadi D, et al. (2019) The Seattle report on database research. SIGMOD Record 48(4):44–53
2. Agrawal P, et al. (2019) Data platform for machine learning. In: SIGMOD
3. Ali M, et al. (2020) PyKEEN 1.0: A python library for training and evaluating knowledge graph embeddings. arXiv preprint arXiv:200714175
4. Angles R, et al. (2018) G-CORE: A core for future graph query languages. In: SIGMOD
5. Baylor D, et al. (2017) TFX: A TensorFlow-based production-scale machine learning platform. In: SIGKDD
6. Belleau F, Nolin MA, Tourigny N, Rigault P, Morissette J (2008) Bio2RDF: Towards a mashup to build bioinformatics knowledge systems. Journal of Biomedical Informatics 41
7. Bordes A, Usunier N, Garcia-Duran A, Weston J, Yakhnenko O (2013) Translating embeddings for modeling multi-relational data. In: NIPS
8. Costabello L, et al. (2019) AmpliGraph: A library for representation learning on knowledge graphs. URL <https://doi.org/10.5281/zenodo.2595043>
9. Dasu T, Johnson T (2003) Exploratory Data Mining and Data Cleaning. John Wiley & Sons
10. Davis R, Shrobe H, Szolovits P (1993) What is a knowledge representation? AI Magazine 14:17–17
11. Dettmers T, Minervini P, Stenetorp P, Riedel S (2018) Convolutional 2d knowledge graph embeddings. In: AAAI

12. Doan A (2018) Human-in-the-loop data analysis: A personal perspective. In: Proc. Workshop on Human-In-the-Loop Data Analytics (HILDA)
13. Dong XL (2018) Challenges and innovations in building a product knowledge graph. In: SIGKDD
14. D'silva JV, De Moor F, Kemme B (2018) AIDA - Abstraction for advanced in-database analytics. PVLDB 11
15. Garcia-Molina H, Ullman JD, Widom J (2008) Database Systems: The Complete Book, 2nd edn. Pearson
16. Giles CL, Bollacker KD, Lawrence S (1998) CiteSeer: An automatic citation indexing system. In: ACM DL
17. Govind Y, et al. (2019) Entity matching meets data science: A progress report from the Magellan project. In: SIGMOD
18. Haase P, Broekstra J, Eberhart A, Volz R (2004) A comparison of RDF query languages. In: ISWC
19. Hagedorn S, Kläbe S, Sattler KU (2021) Putting Pandas in a box. In: CIDR
20. Han X, et al. (2018) OpenKE: An open toolkit for knowledge embedding. In: EMNLP
21. Jenatton R, Roux NL, Bordes A, Obozinski GR (2012) A latent factor model for highly multi-relational data. In: NIPS
22. Jindal A, et al. (2021) Magpie: Python at speed and scale using cloud backends. In: CIDR
23. Kaminski M, Kostylev EV, Cuenca Grau B (2016) Semantics and expressive power of subqueries and aggregates in SPARQL 1.1. In: WWW
24. Kochut K, Janik M (2007) SPARQLer: Extended SPARQL for semantic association discovery. In: ESWC
25. Lehmann J, et al. (2015) DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. Semantic Web 6
26. Matsumoto S, Yamanaka R, Chiba H (2018) Mapping RDF graphs to property graphs. In: Proc. Joint Int. Semantic Technology Conf. (JIST)
27. McCallum AK, Nigam K, Rennie J, Seymore K (2000) Automating the construction of Internet portals with machine learning. Information Retrieval 3(2)
28. Mohamed A, Abuoda G, Ghanem A, Kaoudi Z, Aboulnaga A (2020) RDFFrames: Knowledge graph access for machine learning tools. PVLDB 13, (Demonstration)
29. Navigli R, Ponzetto SP (2012) Babelnet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. Artificial Intelligence 193
30. Nguyen DQ (2017) An overview of embedding models of entities and relationships for knowledge base completion. arXiv preprint arXiv::170308098
31. Nickel M, Tresp V, Kriegel HP (2011) A three-way model for collective learning on multi-relational data. In: ICML
32. Nickel M, Murphy K, Tresp V, Gabrilovich E (2015) A review of relational machine learning for knowledge graphs. Proc IEEE 104
33. Nickel M, Rosasco L, Poggio T (2016) Holographic embeddings of knowledge graphs. In: AAAI
34. Pérez J, Arenas M, Gutiérrez C (2010) nSPARQL: A navigational language for RDF. J Web Semantics 8
35. Petersohn D, et al. (2020) Towards scalable dataframe systems. PVLDB 13
36. Pirahesh H, Hellerstein JM, Hasan W (1992) Extensible/rule based query rewrite optimization in Starburst. In: SIGMOD
37. Pujara J, Augustine E, Getoor L (2017) Sparsity and noise: Where knowledge graph embeddings fall short. In: EMNLP
38. Rebele T, et al. (2016) YAGO: A multilingual knowledge base from Wikipedia, Wordnet, and Geonames. In: ISWC
39. Schwarte A, Haase P, Hose K, Schenkel R, Schmidt M (2011) FedX: Optimization techniques for federated query processing on linked data. In: ISWC
40. Sculley D, et al. (2015) Hidden technical debt in machine learning systems. In: NIPS
41. Suchanek FM, Kasneci G, Weikum G (2007) YAGO: A core of semantic knowledge. In: WWW
42. Vrandečić D (2012) Wikidata: A new platform for collaborative data collection. In: WWW
43. Wang Q, Mao Z, Wang B, Guo L (2017) Knowledge graph embedding: A survey of approaches and applications. TKDE 29
44. Wang Y, Gemulla R, Li H (2018) On multi-relational link prediction with bilinear models. In: AAAI
45. West R, et al. (2014) Knowledge base completion via search-based question answering. In: WWW
46. Wickham H (2011) The split-apply-combine strategy for data analysis. J Statistical Software 40
47. Zaharia M, et al. (2016) Apache Spark: A unified engine for big data processing. CACM 59
48. Zaharia M, et al. (2018) Accelerating the machine learning lifecycle with MLflow. IEEE Data Eng Bull 41

A Full Python Code for Case Studies

A.1 Movie Genre Classification

```

# RDFSFrames imports, graph, and prefixes
from rdfframes.knowledge_graph import KnowledgeGraph
from rdfframes.dataset.rdfpredicate import RDFSPredicate
from rdfframes.utils.constants import JoinType
from rdfframes.client.http_client import HttpClientDataFormat, HttpClient
graph = KnowledgeGraph(graph_uri='http://dbpedia.org',
                       prefixes= {'dcterms': 'http://purl.org/dc/terms/',
                                  'rdfs': 'http://www.w3.org/2000/01/rdf-schema#',
                                  'dbpprop': 'http://dbpedia.org/property/',
                                  'dbpr': 'http://dbpedia.org/resource/'})

# RDFSFrames code for creating the dataframe
dataset = graph.feature_domain_range('dbpp:starring','movie', 'actor')
dataset = dataset.expand('actor',[(('dbpp:birthPlace', 'actor_country'),('rdfs:label', 'actor_name')])
    .expand('movie', [(('rdfs:label', 'movie_name'),('dcterms:subject', 'subject'),
                        ('dbpp:country', 'movie_country'),('dbpo:genre', 'genre', Optional))].cache())
american = dataset.filter({'actor_country': ['regex(str(?actor_country),"USA")']})
prolific = dataset.group_by(['actor']).count('movie', 'movie_count', unique=True).filter({'movie_count': ['>=100']})
movies = american.join(prolific,'actor', OuterJoin).join(dataset, 'actor', InnerJoin)

# Client and execution
output_format = HttpClientDataFormat.PANDAS_DF
client = HttpClient(endpoint_url=endpoint, return_format=output_format)
df = movies.execute(client, return_format=output_format)

# Preprocessing and preparation
import re
import nltk

def clean(dataframe):
    for i, row in df.iterrows():
        if df.loc[i]['genre'] != None:
            value =df.at[i, 'genre']
            if re.match(regex,str(value)) is not None:
                df.at[i, 'genre'] = value.split('/')[1]
    return dataframe

# Remove URL from the 'genre' and convert to label keys
df=clean(df)

# Find the most most frequent genres
all_genres = nltk.FreqDist(df['genre'].values)
all_genres_df = pd.DataFrame({'genre':list(all_genres.keys()), 'Count':list(all_genres.values())})
all_genres_df.sort_values(by=['Count'],ascending=False)

# In this example, use 900 movies as a cut off for the frequent movies
most_frequent_genres = all_genres_df[all_genres_df['Count']> 900]
df = df[df['genre'].isin(list(most_frequent_genres['genre']))]

# Features and factorization
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

df= df.apply(lambda col: pd.factorize(col, sort=True)[0])
features = ["movie_name", "actor_name", "actor_country","subject","movie_country", "subject"]
df = df.dropna(subset=['genre'])
x = df[features]
y = df['genre']
x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=20)
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.fit_transform(x_test)

# Random Forest classifier
from sklearn.ensemble import RandomForestClassifier

model=RandomForestClassifier(n_estimators=100)
model.fit(x_train,y_train)
model.fit(x_train,y_train)
y_pred=clf.predict(x_test)
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

```

Listing 12: Full code for movie genre classification.

A.2 Topic Modeling

```

# RDFFrames imports, graph, prefixes, and client
import pandas as pd
from rdfframes.client.http_client import HttpClientDataFormat, HttpClient
from rdfframes.knowledge_graph import KnowledgeGraph
graph = KnowledgeGraph(
    graph_uri = 'http://dblp.l3s.de',
    prefixes = {"xsd": "http://www.w3.org/2001/XMLSchema#",
                "swrc": "http://swrc.ontoware.org/ontology#",
                "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
                "dc": "http://purl.org/dc/elements/1.1/",
                "dcterm": "http://purl.org/dc/terms/",
                "dblp": "http://dblp.l3s.de/d2r/resource/conferences/"})
output_format = HttpClientDataFormat.PANDAS_DF
client = HttpClient(endpoint_url=endpoint, port=port, return_format=output_format)

# RDFFrames code for creating the dataframe
papers = graph.entities('swrc:InProceedings', paper)
papers = papers.expand('paper', [(dc:creator', 'author'), (dcterm:issued', 'date'), ('swrc:series', 'conference'),
                                ('dc:title', 'title')]).cache()
authors = papers.filter({'date': ['>=2005'], 'conference': ['In(dblp:vldb,dblp:sigmod)']}).group_by(['author'])
    .count('paper', 'n_papers').filter({'n_papers': '>=20', 'date': ['>=2005']})
titles = papers.join(authors, 'author', InnerJoin).select_cols(['title'])
df = titles.execute(client, return_format=output_format)

# Preprocessing and cleaning
from nltk.corpus import stopwords
df['clean_title'] = df['title'].str.replace("[^a-zA-Z#]", "_")
df['clean_title'] = df['clean_title'].apply(lambda x: x.lower())
df['clean_title'] = df['clean_title'].apply(lambda x: '_'.join([w for w in str(x).split() if len(w)>3]))
stop_words = stopwords.words('english')
tokenized_doc = df['clean_title'].apply(lambda x: x.split())
df['clean_title'] = tokenized_doc.apply(lambda x:[item for item in x if item not in stop_words])

# Vectorization and SVD model using the scikit-learn library
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD
vectorizer = TfidfVectorizer(stop_words='english', max_features= 1000, max_df = 0.5, smooth_idf=True)
Tfidf_title = vectorizer.fit_transform(df['clean_title'])
svd_model = TruncatedSVD(n_components=20, algorithm='randomized', n_iter=100, random_state=122)
svd_model.fit(Tfidf_titles)

# Extracting the learned topics and their keyterms
terms = vectorizer.get_feature_names()
for i, comp in enumerate(svd_model.components_):
    terms_comp = zip(terms, comp)
    sorted_terms = sorted(terms_comp, key= lambda x:x[1], reverse=True)[:7]
    print_string = "Topic"+str(i)+":_:"
    for t in sorted_terms:
        print_string += t[0] + "_"

```

Listing 13: Full code for topic modeling.

A.3 Knowledge Graph Embedding

```

# Get all triples where the object is a URI
from rdfframes.knowledge_graph import KnowledgeGraph
from rdfframes.dataset.rdfpredicate import RDFPredicate
from rdfframes.client.http_client import HttpClientDataFormat, HttpClient
output_format = HttpClientDataFormat.PANDAS_DF
client = HttpClient(endpoint_url=endpoint, port=port, return_format=output_format, timeout=timeout,
                    default_graph_uri=default_graph_url, max_rows=max_rows)
dataset = graph.feature_domain_range(s, p, o).filter({'o': ['isURI']})
df = dataset.execute(client, return_format=output_format)

# Train/test split and create ComplEx model from ampligraph library
from ampligraph.evaluation import train_test_split_no_unseen
triples = df.to_numpy()
X_train, X_test = train_test_split_no_unseen(triples, test_size=10000)
from ampligraph.latent_features import ComplEx
from ampligraph.evaluation import evaluate_performance, mrr_score, hits_at_n_score
model = ComplEx(batches_count=50, epochs=300, k=100, eta=20, optimizer='adam', optimizer_params={'lr':1e-4},
               loss='multiclass_nll', regularizer='LP', regularizer_params={'p':3, 'lambda':1e-5}, seed=0, verbose=True)
model.fit(X_train)
# Evaluate embedding model
filter_triples = np.concatenate((X_train, X_test))
ranks = evaluate_performance(X_test, model=model, filter_triples=filter_triples,
                             use_default_protocol=True, verbose=True)

mr = mrr_score(ranks)
mrr = mrr_score(ranks)

```

Listing 14: Full code for knowledge graph embedding.

B Description of Queries in the Synthetic Workload

Table 3: Description of the queries in the syntnthetic workload.

Query	English Description	RDFFrames Operators	SPARQL Features
Q1	Get a list of films in DBpedia. For each film, return the actor, language, country, genre, story, and studio, in addition to the director, producer, and title (if available).	expand (including optional predicates)	OPTIONAL, DISTINCT
Q2	Get a list of actors available in the DBpedia or YAGO graphs.	join (outer) between two graphs, filter	OPTIONAL, FILTER, UNION
Q3	Get a list of American actors available in both the DBpedia and YAGO graphs.	join (inner) between two graphs, expand, filter	FILTER
Q4	Get the nationality, place of birth, and date of birth of each basketball player in DBpedia, in addition to the sponsor, name, and president of his team (if available).	join (left outer) between two expandable datasets, expand (including optional predicates)	OPTIONAL
Q5	Get the players (athletes) in DBpedia and their teams, group by teams, count players, and expand the team's name.	group.by, count, expand	GROUP BY, COUNT, DISTINCT
Q6	For films in DBpedia that are produced by any studio in India or the United States excluding 'Eskay Movies', and that have one of the following genres (film score, soundtrack, rock music, house music, or dubstep), get the actor, director, producer, time and language.	expand, filter	FILTER
Q7	For the films in DBpedia, get actors, director, country, producer, language, title, genre, story, and studio. Filter on country, studio, genre, and runtime.	expand, filter	FILTER
Q8	Get the nationality, place of birth, and date of birth of each basketball player in DBpedia, in addition to the sponsor, name, and president of his team.	join (inner) between two expandable datasets, expand	Multiple conjunctive graph patterns
Q9	Get the list of basketball players in DBpedia, their teams, and the number of players on each team.	group.by, count, expand	GROUP BY, COUNT, DISTINCT
Q10	For films in DBpedia that are produced by any studio in India or the United States excluding 'Eskay Movies', and that have one of the following genres (film score, soundtrack, rock music, house music, or dubstep), get the actor and language, in addition to the producer, director, and title (if available).	expand (including optional predicates), filter	OPTIONAL, FILTER
Q11	Get the list of athletes in DBpedia. For each athlete, return his birthplace and the number of athletes who were born in that place.	group.by, count, expand	GROUP BY, COUNT, DISTINCT
Q12	Get the pairs of films in DBpedia that belong to the same genre and are produced in the same country. For each film in each pair, return the actor, country, story, language, genre, and studio, in addition to the director, producer, and title (if available).	group.by on multiple columns, count, expand (including optional predicates)	GROUP BY, COUNT, OPTIONAL, DISTINCT
Q13	Get the sponsor, name, president, and the number of basketball players of each basketball team in DBpedia.	join (inner) between two datasets (expandable, group.by)	GROUP BY, COUNT, DISTINCT
Q14	Get the sponsor, name, president, and the number of basketball players (if available) of each basketball team in DBpedia.	join (left outer) between two datasets (expandable, group.by), expand (including optional predicates)	GROUP BY, COUNT, OPTIONAL, DISTINCT
Q15	Get a list of the books in DBpedia that were written by American authors who wrote more than two books. For each author, return the birth place, country, and education, and for each book return the title, subject, country (if available), and publisher (if available).	join (outer), group.by, count, expand (including optional predicates), filter	GROUP BY, COUNT, HAVING, OPTIONAL, FILTER, UNION
Q16	Get a list of people in the DBpedia graph who were born in the United States. Get a list of authors from the DBLP graph who have publications dated after 2015. Get a list of people in the YAGO graph who are citizens of the United States. Join the three lists retrieved from the three graphs on name.	join (full outer) between three graphs, expand (including optional predicates), filter	OPTIONAL, FILTER, DISTINCT, UNION

C Naive SPARQL Query for Movie Genre Classification

```

PREFIX dbpp: <http://dbpedia.org/property/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpo: <http://dbpedia.org/ontology/>
PREFIX dbpr: <http://dbpedia.org/resource/>
SELECT DISTINCT ?actor_name ?movie_name ?actor_country ?subject ?genre
FROM <http://dbpedia.org> WHERE
  {{{ SELECT * WHERE
    {{ SELECT * WHERE
      { { SELECT ?movie ?actor WHERE
          { ?movie dbpp:starring ?actor } }
        { SELECT ?actor ?actor_country WHERE
            { ?actor dbpp:birthPlace ?actor_country } }
        { SELECT ?actor ?actor_name WHERE
            { ?actor rdfs:label ?actor_name } }
        { SELECT ?movie ?movie_name WHERE
            { ?movie rdfs:label ?movie_name } }
        { SELECT ?movie ?subject WHERE
            { ?movie dcterms:subject ?subject } }
        { SELECT ?movie ?movie_country WHERE
            { ?movie dbpp:country ?movie_country } }
        { SELECT ?actor ?actor_country WHERE
            { ?actor dbpp:birthPlace ?actor_country
              FILTER regex(str(?actor_country), "USA") } }
        { SELECT ?movie ?genre WHERE

```

```

        { OPTIONAL
          { ?movie dbpo:genre ?genre } } } } }
OPTIONAL
  { SELECT DISTINCT ?actor (COUNT(DISTINCT ?movie) AS ?movie_count) WHERE
    { { SELECT ?movie ?actor WHERE
      { ?movie dbpp:starring ?actor } }
      { SELECT ?actor ?actor_country WHERE
        { ?actor dbpp:birthPlace ?actor_country } }
      { SELECT ?actor ?actor_name WHERE
        { ?actor rdfs:label ?actor_name } }
      { SELECT ?movie ?movie_name WHERE
        { ?movie rdfs:label ?movie_name } }
      { SELECT ?movie ?subject WHERE
        { ?movie dcterm:subject ?subject } }
      { SELECT ?movie ?movie_country WHERE
        { ?movie dbpp:country ?movie_country } }
      { SELECT ?movie ?genre WHERE
        { OPTIONAL
          { ?movie dbpo:genre ?genre } } } } } }
    GROUP BY ?actor
    HAVING ( COUNT(DISTINCT ?movie) >= 100 ) } } }
UNION
  { SELECT * WHERE
    { { SELECT DISTINCT ?actor (COUNT(DISTINCT ?movie) AS ?movie_count) WHERE
      { { SELECT ?movie ?actor WHERE
        { ?movie dbpp:starring ?actor } }
        { SELECT ?actor ?actor_country WHERE
          { ?actor dbpp:birthPlace ?actor_country } }
        { SELECT ?actor ?actor_name WHERE
          { ?actor rdfs:label ?actor_name } }
        { SELECT ?movie ?movie_name WHERE
          { ?movie rdfs:label ?movie_name } }
        { SELECT ?movie ?subject WHERE
          { ?movie dcterm:subject ?subject } }
        { SELECT ?movie ?movie_country WHERE
          { ?movie dbpp:country ?movie_country } }
        { SELECT ?movie ?genre WHERE
          { OPTIONAL { ?movie dbpo:genre ?genre } } } } } }
      GROUP BY ?actor
      HAVING ( COUNT(DISTINCT ?movie) >= 100 ) }
    OPTIONAL
      { SELECT * WHERE
        { { SELECT ?movie ?actor WHERE
          { ?movie dbpp:starring ?actor } }
          { SELECT ?actor ?actor_country WHERE
            { ?actor dbpp:birthPlace ?actor_country } }
          { SELECT ?actor ?actor_name WHERE
            { ?actor rdfs:label ?actor_name } }
          { SELECT ?movie ?movie_name WHERE
            { ?movie rdfs:label ?movie_name } }
          { SELECT ?movie ?subject WHERE
            { ?movie dcterm:subject ?subject } }
          { SELECT ?movie ?movie_country WHERE
            { ?movie dbpp:country ?movie_country } }
          { SELECT ?actor ?actor_country WHERE
            { ?actor dbpp:birthPlace ?actor_country
              FILTER regex(str(?actor_country), "USA") } }
          { SELECT ?movie ?genre WHERE
            { OPTIONAL { ?movie dbpo:genre ?genre } } } } } } } } } } } } } } }
  }

```

Listing 15: Naive SPARQL query corresponding to the SPARQL query shown in Listing 7.

D Naive SPARQL Query for Topic Modeling

```

SELECT ?title
FROM <http://dblp.13s.de> WHERE
  { {SELECT ?paper WHERE {?paper rdf:type swrc:InProceedings}}.
    {SELECT ?paper ?author WHERE {?paper dc:creator ?author}}.
    {SELECT ?paper ?date WHERE {?paper dcterm:issued ?date }}.
    {SELECT ?paper ?conference WHERE {?paper swrc:series ?conference}} .
    {SELECT ?paper ?title WHERE {?paper dc:title ?title}}.
    {SELECT ?paper ?date WHERE {?paper dcterm:issued ?date FILTER ( year(xsd:dateTime(?date)) >= 2005 ) } }
  }
  { SELECT ?author WHERE
    { { SELECT ?author COUNT(?paper) as ?count_paper WHERE
      { {SELECT ?paper WHERE {?paper rdf:type swrc:InProceedings}}.
        {SELECT ?paper ?author WHERE {?paper dc:creator ?author}}.
        {SELECT ?paper ?date WHERE {?paper dcterm:issued ?date }}.
        {SELECT ?paper ?conference WHERE {?paper swrc:series ?conference}} .
        {SELECT ?paper ?title WHERE {?paper dc:title ?title}}.
        {SELECT ?paper ?date WHERE {?paper dcterm:issued ?date FILTER ( year(xsd:dateTime(?date)) >= 2000 ) } } .
        {SELECT ?paper ?conference WHERE {?paper swrc:series ?conference FILTER( ?conference IN (dblp:svldb, dblp:sigmod))}}
      }
      GROUP BY ?author }
    FILTER ( ?count_paper >= 20 ) } } }

```

Listing 16: Naive SPARQL query corresponding to the SPARQL query shown in Listing 9.