

RESEARCH ARTICLE

maplib: Interactive, Literal RDF Model Mapping for Industry

MAGNUS BAKKEN Department of Computer Science, Norwegian University of Science and Technology—NTNU, 2815 Gjøvik, Norway
Prediktor AS, 1630 Fredrikstad, Norway

e-mail: magba@stud.ntnu.no

This work was supported by the Research Council of Norway and Prediktor AS through the Industrial Ph.D. Scheme under Grant 316656.

ABSTRACT Knowledge graphs are important for industrial digitalization. Industrial knowledge graphs are often mapped from multiple existing large data sources, and creating a mapping requires the time of scarce subject matter experts (SME). Interactive, literal programming for large scale mapping would allow mapping engineers to make good use of SME time, and document their work. Currently, there are no open-source tools supporting such a process. To solve this problem, we implement maplib, which leverages existing tooling from data science. In data science, there is widespread use of literate programming using frameworks such as Jupyter notebooks to interactively prepare data and create analyses using in-memory tables called DataFrames. Maplib is implemented in Rust using Polars DataFrames and has Python bindings, allowing us to leverage tooling used in data science. Maplib implements the OTTR mapping language, which is highly suited for industrial use cases. Maplib features a SPARQL engine defined directly on DataFrames, making querying possible immediately after mapping. We evaluate our approach by comparing mapping and querying performance with Morph-KGC and SPARQL Anything on the GTFS Madrid benchmark. Our approach materializes the graph and is ready to query $47\times-182\times$ faster, and scales to models that are over twice as large. Morph-KGC and SPARQL Anything perform better for most, but not all of the queries once the graph has been constructed. On the end-to-end task of mapping and querying however, which is very important for interactive mapping, maplib performs better for most queries.


INDEX TERMS Knowledge graph construction, knowledge graph mapping, RDF mapping.

I. INTRODUCTION

Knowledge graphs are digital representations of our knowledge of the world made up of nodes representing concrete or abstract entities or concepts, as well as edges between them, representing relationships of different kinds. In industrial settings, knowledge graphs can integrate heterogeneous data sources into a unified model of an asset or process. Industrial knowledge graphs are important for industrial digitalization [1], [2]. Constructing or mapping industrial knowledge graphs requires the time of scarce subject matter experts (SME). Interactive, literal approaches to knowledge graph construction can be useful in these settings, as the short

duration of the feedback loop allows an engineer to make good use of the SME.

Data analysis and data engineering in Python is very popular. In combination with interactive Jupyter Notebooks [3], it is thought to improve the speed with which data professionals can explore, process and analyse a data set. Literal programming was invented by Donald Knuth, and is a form of programming which follows the form of literature, the flow of the program follows a line of reasoning [4]. In literal programming, prose is interspersed with code and outputs. Jupyter Notebooks are considered a form of literal programming, and allows the data analysis process to be inspected and documented as it proceeds [5], [6]. Data analysis and data engineering in Python very often involves the Pandas library [7], which has powerful in-memory tables called DataFrames. Pandas offers a rich and performant set

The associate editor coordinating the review of this manuscript and approving it for publication was Mu-Yen Chen .

of operations on these DataFrames, and can import from a myriad of heterogeneous sources with few lines of code [8]. These operations overlap to a large degree with the tasks that are performed when constructing a knowledge graph.

Open source, interactive, literal knowledge graph construction tools that allow intermediary results to be inspected and manipulated do not exist, making it difficult to quickly and exploratively create knowledge graphs. Integrating with the DataFrame-based Jupyter Notebook ecosystem could provide this support with no additional work. While there does exist an open-source tool for knowledge graph construction and querying in Python called Morph-KGC [9] which is based on Pandas, it relies on atomic, non-interactive execution of knowledge graph construction, and does not actually integrate with DataFrames. This lacking integration is likely due to the use of end-to-end declarative mappings using RML, which is a mapping language that defines end-to-end transformations from sources such as databases or files to knowledge graphs [10]. RML implementations typically allow no inspection and no interactivity with intermediary results [11]. RML also does not support templates, which are important in industrial settings with a large degree of modularity [12]. Morph-KGC relies on third-party databases and performs a data transformation and copying operation in order to populate these databases with the mapped knowledge graph [13]. Using DataFrames as a database primitive, one can actually forego such a transformation step and query the mapping output directly.

To address these gaps, we introduce maplib. Maplib is written in Rust, and uses Polars DataFrames [14] as the core data structure. Polars DataFrames perform better than Pandas DataFrames due to lazy execution, high parallelism, and its implementation in Rust [14], [15]. It has Python bindings that allow DataFrames to be transferred back and forth between Rust and Python with minimal cost. Maplib implements OTTR [12], which is a template-based mapping language oriented around API-like functional abstraction. These abstractions are highly suitable for integration with DataFrames. The templates-based nature of OTTR also makes it highly suitable for industrial settings. We adapt a DataFrames-based SPARQL query engine that we developed earlier [16] for maplib, and make it possible to query mapped knowledge graphs immediately after they are constructed, without a separate transformation/copying step. The SPARQL engine of maplib is also able to extend the knowledge graph with the results of the construct query without transformation or copying, permitting fast enrichment of the knowledge graph. To evaluate maplib, we compare it with state-of-the-art solutions Morph-KGC [9] and SPARQL Anything [17] on the Madrid GTFS benchmark [18], which features knowledge graph construction together with challenging queries.

Our paper is structured as follows. We first introduce relevant background knowledge in Section II. We consider the motivation, research problem, and requirements in Section III. We describe existing solutions in relation to our

requirements in Section IV. The solution approach and implementation is described in Section V. We evaluate maplib in Section VI, before presenting our conclusions as well as future work in Section VII.

II. BACKGROUND

This section covers background knowledge important to our work. We discuss Semantic Web Technologies (SWT) and knowledge graphs in Section II-A. Major languages used to construct knowledge graphs are introduced in Section II-B. We introduce literate programming and how it is being used in modern data analysis in Section II-C. Finally, we describe important technologies for modern data analysis and data engineering that we use in this work in Section II-D.

A. SEMANTIC WEB TECHNOLOGIES AND KNOWLEDGE GRAPHS

The Semantic Web started as an initiative to create interoperable ways of sharing meaningful information on the Web, where websites would share information in particular formats, allowing information to be integrated across websites and used to support new use cases [19]. Out of this work came powerful ways of representing knowledge, and for querying and reasoning about that knowledge.

The Resource Description Framework (RDF) is a way of representing knowledge based on triples containing subjects, verbs, and objects [20]. Triples either describe some relationship between the subject and the object or associate a piece of data with the subject. The subjects and objects of such a collection of triples form the nodes of a graph, while the verbs form the edges. The non-data nodes are called resources and may refer to real or imagined entities. Verbs typically have agreed-upon meanings. In sum, such a graph represents knowledge about a domain of discourse, and is often referred to as a knowledge graph.

Higher-order knowledge graphs called ontologies represent the concepts and terms in the knowledge graph, and are typically used to represent a particular domain of discourse in a uniform and coherent way [21]. Knowledge graphs based on RDF can be queried with SPARQL [22], a query language which allows users to formulate queries using the concepts from ontologies. We call such data access Ontology Based Data Access (OBDA) [23]. Compared to query languages such as SQL, SPARQL abstracts away more technical details on how data are stored, and focuses on expressing the conceptual relationships inherent in our information retrieval task. SPARQL can either be executed on a triple store which physically stores the RDF triples, or we can have data stay in an existing database and translate SPARQL queries into the query language of this database. In the first case, we say that we are materializing the Knowledge Graph. In the latter case, we say that we have a Virtual Knowledge Graph (VKG) [24].

B. RDF MAPPING LANGUAGES

A core challenge is to transform data from other formats into RDF-based knowledge graphs so that it can be e.g. queried and reasoned about. The Relational database To RDF Mapping Language (R2RML) is a W3C-recommended declarative language for describing mappings from relational databases to RDF [25]. The language describes how to construct the subject IRI for all rows in a table, and how to construct a number of triples for each row. The language is designed with two use-cases in mind. The first use case is to materialize RDF triples based on data in a relational database. The second use case is to support VKGs. That is, the mapping should contain the information necessary to be able to answer SPARQL queries by translating them into SQL queries over the relational database [26].

RML is a language that extends R2RML to cover other data sources such as CSV and JSON files [10], iterating over rows of the CSV or matches of a JSONPath in case of a JSON file. Multiple implementations of RML exist, such as SDM-RDFizer [27] and Morph-KGC [9] for Python. R2RML has also been extended (R2RML-F) with a facility for specifying functions that should be applied to the input data before it is used to instantiate triples [28], using a standardized set of functions called FnO [29]. Since RML includes R2RML, we will use the term “RML” to refer to them collectively in the rest of the paper.

The Façade-X mapping language allows users to introduce heterogeneous data sources such as relational databases, JSON files, and CSV files into SPARQL through the SPARQL Service construction [17]. An abstraction called a *facade* is used to expose such data as RDF, using special predicates that are used to access, e.g., the columns of a CSV file or keys in a JSON. This allows end-to-end mappings to be defined declaratively, using lightly annotated SPARQL queries.

However, both RML and Façade-X lack facilities for functional abstraction over mappings. Individual components may be reused, but it is impossible to group and nest such components. The issue of reusable components has been addressed by Ontology Design Patterns (ODP) [30], but has been criticized for being hard to use in practice [31]. In particular, ODP was criticized for not having user-facing abstractions [31]. The Reasonable Ontology Templates (OTTR) mapping language addresses this shortcoming by creating a templating language that is based on functional templates with arguments [12].

OTTR templates are essentially reusable macros that can be referenced by users. The head of an OTTR macro consists of its parameters. In the body, other templates are called using a combination of these parameters and static terms. Executing an OTTR template consists of rewriting each template in the body, replacing each parameter with the argument provided. The process of substitution is repeated with the templates in the body until there are only `ottr:Triple`-templates, which have three arguments that map one-to-one with the subject, verb, and object of the RDF triples to be created.

OTTR templates are nested, but cycles are disallowed [31]. We discuss OTTR in greater detail and provide an example in Section V.

The Terse Syntax for OTTR (stOTTR) is a syntax for specifying OTTR templates in text in a way that seeks to be easy to use [12]. OTTR differs significantly from RML and Façade-X in that it makes no attempt to describe the data source or to allow translation of SPARQL queries to a source database. OTTR is especially suited for industrial use cases, as these tend to be built on standardized, modular representations that are combined in myriad ways to represent, e.g., various types of equipment and their properties [12], [32], [33].

C. LITERATE PROGRAMMING

Literate programming was invented by Donald Knuth, and refers to programming that takes a form similar to literature [4]. The purpose of a literate program is to structure code for readability, following a train of thought, motivating and describing the steps with prose. In a step called weaving, the output of the program is combined with the verbal description to create an integrated document [4].

Modern literate programming environments such as Jupyter Notebook have become very popular among data scientists for documenting data processing and analysis, particularly in an exploratory stage [5]. Jupyter Notebooks combine literate programming with interactivity, allowing data scientists to get rapid feedback at each stage of data processing and analysis, provided these steps are not long-running.

D. PANDAS, DataFrames, ARROW, POLARS AND PARQUET

Pandas is a high-performance open-source library for data manipulation and analysis in Python [7], which has become a standard tool in data analysis. In a survey of over 23,000 Python developers conducted by the Python Software Foundation and IDE-vendor JetBrains, 53% of respondents answered that they used Python for data analysis, and 55% of respondents used Pandas. Python is among the world’s most popular programming languages, perhaps the most popular [6]. It is not clear from the survey how representative it is for the population of Python developers, but it is safe to assume that Pandas is an extremely popular data analysis/data engineering tool.

A central data structure in Pandas is the DataFrame. DataFrames are in-memory tables of data, consisting of named columns of data of a homogenous type. In Pandas, DataFrames are built on data structures from NumPy, a Python library for numerical computations [34]. The Apache Arrow project seeks to standardise such columnar in-memory representations, making it easy to share them among processes on the same machine [35]. Agreeing on the standard, one application can pass an Apache Arrow array to another by providing the pointer to in-memory data instead of copying data. Additionally, Apache Arrow Flight enables transport of columnar data from e.g. databases without having

to perform costly row-oriented serialization associated with ODBC [36].

The open-source Polars library is based on Apache Arrow. It is built in Rust, but has Python bindings. It provides an API that is very similar to Pandas DataFrames, but that additionally supports lazy execution, allowing optimizations to be applied to the execution plan. Polars DataFrames can easily be created from Pandas DataFrames and vice versa. Apache Parquet is a column-oriented file format that is commonly used in data lakes [37]. Parquet can be read to- and written from Apache Arrow with very little serialization overhead.

III. PROBLEM

Exploratory data engineering for analytics relies on literate, interactive programming in the notebook format. At their best, notebooks enable data engineers to quickly and collaboratively prototype pipelines for data processing in the form of an illustrated story. Jupyter Notebooks face issues with testability, reproducibility, and modularity that makes it challenging to use them in production and maintain them over time [38], [39], [40]. We will assume that notebooks are rewritten into structured form after the exploration phase is done in order to meet common quality requirements for production code (cf. [38]). The core problem addressed by this work is to provide such a facility for mapping engineers that are building industrial knowledge graphs, which can be used in the exploration phase, and then refactored into structured form for production deployment and ease of maintenance.

In the rest of this section, we will motivate the problem of template-based interactive mapping that integrates well with DataFrames, arguing that such tooling can improve knowledge graph mapping practices, especially for industry. Finally, we describe our requirements.

A. MOTIVATION

1) THE VALUE OF INTERACTIVITY AND LITERAL PROGRAMMING

OBDA is used to make it easier to access data, especially when those data are located in separate databases, as we are able to abstract away the technical schemas of the individual data sets and integrate them in a common RDF model. Mapping engineers need to understand these data sets in order to map them to RDF and integrate them. Attempting to perform a rudimentary data integration is a way of accomplishing this task that provides quick feedback on the assumptions of the mapping engineer.

When integrating enterprise data sets, one can encounter insufficiently documented, ambiguous data that are hard to reconcile without extensive domain knowledge [41]. In such cases, a SME may have to be consulted. Such individuals are typically in high demand, and it is important to make effective use of their time. A similar argument also applies to the target ontology which will be instantiated. In industrial settings, there are sometimes precise modeling rules to follow when creating a model according to a standard. Correct

instantiation of such ontologies relies on domain knowledge. Being able to interactively work on constructing a mapping is helpful, as this provides immediate feedback to the mapping engineer on how well she has understood and manipulated the source data sets, and on how appropriate the mapping is. An interactive mapping environment should enable the mapping engineer to quickly follow up with the SME to resolve questions.

The form of the feedback is important in order to be able to collaborate and communicate the results of the exploratory data integration. It is ineffective to manually collect the outputs of various mapping scripts and programs in a document. Corrections to the mapping upstream to such outputs lead to repetitive manual updating of the documentation. A literal programming environment such as Jupyter Notebooks avoids this toil, as the script is in a literal form with outputs that update when the script is rerun. The audience of this literal program is not just mapping engineers. A literal mapping will inform the SME on how the data sets of the organization can be integrated, and make visible the consequences of such integration through queries or other means of inspection.

2) THE VALUE OF INTEGRATING WITH EXISTING DATA ENGINEERING TOOLS AND FORMATS

Today, there are very mature data engineering tools that are very well maintained. Pandas is likely to remain so, as thousands of engineers and organizations depend on it. There are multiple advantages to being highly interoperable with DataFrame-based data engineering tooling that are due to the broad acceptance of these tools. It is easy to find documentation, and common issues are well documented on websites such as Stack Overflow. There is extensive support for reading and writing data to and from different file formats and databases. Many of the data preparation tasks that must be conducted by mapping engineers are already implemented. There is support for configurable literate outputs in Jupyter Notebooks. Existing data engineering tooling has high performance enabling interactive processing of large datasets. Many data engineers will know these tools very well, and thus face less effort in adopting a mapping engineering tool as their existing skills can be reused. By integrating with such tooling, it is possible for developers to focus on solving the tasks in mapping engineering that are not found in general data engineering tools. Less effort is required to maintain the tooling over time, as important functionality that is not unique to mapping engineering will be maintained separately by a much larger community.

Increasingly, analytical databases are relying on the column-based Apache Arrow and Apache Parquet formats for storage, transmission, and computations on data [42], [43], [44], [45], [46]. We have already discussed how these formats can drastically reduce serialization overhead. Modern DataFrame-based data engineering tooling has good interoperability with these formats. By interoperating well with DataFrames we can exploit these technological developments

in knowledge graph construction, for instance by reducing the time taken to write and upload a knowledge base into a triplestore – if and when the triplestore supports such formats.

3) THE VALUE OF ENRICHMENT FOR INDUSTRIAL INFORMATION MODELS

We can consider the case where an application uses SPARQL to access a materialized knowledge graph as a read-only cache to offload a transactional database. Such a situation occurs often in data-intensive systems, and can be handled with caches built using batch processing [47]. However, some SPARQL queries are by their very nature expensive, and may in any case be more time-consuming than we would like for our application. In this case, it can be advantageous to materialize auxiliary RDF properties of interest. Such enrichment is obviously supported with SPARQL insert.

Access control is a feature that is not standardized and supported by SPARQL engines. In enterprise settings, it is important to limit access, for instance in accordance with geographical location or with the role of the user. Multiple approaches exist that rely on materializing access control as part of the knowledge graph, and on subsequently rewriting SPARQL queries to ensure the user only accesses the appropriate part of the graph [48], [49]. SPARQL insert queries are one way of creating and maintaining such materialized access control, provided the knowledge graph is sufficiently standardized.

Industrial knowledge graphs may index real-time data arising from sensors and actuators in facilities such as factories or power plants [50]. Applications may read from these values, producing aggregates, KPIs or other derived data, and we have argued in previous work that queries over industrial knowledge graphs can determine the integration of reusable applications [16]. The output data from these applications should also be indexed by the industrial knowledge graph, and an enrichment step is one possible place to add these nodes.

4) THE VALUE OF TEMPLATES

We have already discussed how industrial standards for representing information often are based on components that are composed in order to represent e.g. particular pieces of equipment. For instance, the IEC 61850 standard defines ways of representing information from electrical equipment in part 7-3 [51] that is reused when representing sensor data from hydroelectric power plants in IEC 61850-7-410 [52] and when representing sensor data from wind turbines in IEC 61400-25 [53]. A template-based mapping tool will be able to reuse these templates and create more uniform models. Uniform models are important when knowledge graphs support industrial application integrations, as they make it easier to identify places where e.g. a graphical user interface element or statistical analysis can be reused. This is particularly the case when seeking to reuse the same application across industrial installations or across companies.

B. REQUIREMENTS

Our solution will have to support interactive mapping through a scripting language with library-based integration. Moreover, the solution should be interoperable with DataFrames and Apache Arrow-based data engineering tools. Primarily this requirement applies on the input side. Coupled with interactivity, we should be able to pipe outputs from Arrow-based into the mapping library effectively, without using disk-based integration. These tools allow us to define all of the operations of SPARQL. This further implies that the mapped model can be immediately queriable, or queriable with very little I/O overhead if the database is Arrow-based and external.

We will further require the mapping tool to support SPARQL based inspection of mapped models, enabling inspection in the exploration phase, and validations and tests needed for production deployment in later phases. To support industrial scenarios involving templating, our mapping library should support templates. Additionally, we should be able to use this SPARQL support to post-process the model, enriching it with the outputs of construct-queries. Such support will also be important for industrial scenarios where the knowledge graph supports applications.

All of the mentioned functionality should have high performance for large datasets, as this is important for the mapping to be interactive in a practical way. By choosing an in-memory Arrow model, we are limited by available memory. Scaling the solution to work interactively with big data is outside our scope.

We summarise these requirements below:

- R1 Support interactive mapping in a script-based language
- R2 Interoperability with established Apache Arrow-based data engineering tools
- R3 SPARQL-based inspection
- R4 Templating support
- R5 SPARQL-based post-processing with construct
- R6 High performance for large datasets

IV. EXISTING SOLUTIONS

There are multiple existing interactive mapping tools that warrant discussion. SPARQL Anything implements the Façade-X mappings discussed earlier (Section II-B). SPARQL Anything enables the Façade-X language to be used for the Apache Jena ARQ SPARQL engine [54], [55]. SPARQL Anything supports one-off command line interface (CLI) based mapping as well as an Apache Jena Fuseki-based server for executing mappings [55], [56]. Using the server-executable, SPARQL Anything allows an interactive mapping process in the sense that users can alternate between querying source data, inserting data from these sources, and querying and enriching the resulting knowledge base. Users of the server must however interact with it using the SPARQL Endpoint, e.g. using HTTP with JSON or a SPARQL GUI [57]. Morph-CSV is a Python-based tool for executing SPARQL queries over CSV files mapped with RML [58]. The tool is not interactive, as it is run from the CLI.

Development on the Morph-CSV project appears to have stopped, with no new releases since 2020 [59], and the effect of time has rendered us unable to run the Morph-CSV code without considerable debugging of dependencies. We have therefore not considered Morph-CSV any further. Arenas-Guerrero et al. construct a related tool called Morph-KGC in Python which allows users to construct a materialized knowledge graph based on an RML mapping [9], which is then loaded into either RDFLib [60] or Oxigraph [61] for interactive querying using Python. Morph-KGC relies on the data engineering library Pandas to read inputs specified in RML mappings, but cannot directly read from Pandas DataFrames. RML mapping rules are potentially interdependent, and parallelization of their execution is not trivial. Arenas-Guerrero et al. introduce a mapping partitioning approach for RML, which allows for improved execution times and reduced memory usage [9].

In a recent review of declarative RDF generation tools, Van Assche et al. find that existing mapping tools execute the mapping process atomically from source to RDF [11]. This is for instance the case for the above mapping tools. Atomic execution is also the case for existing mapping tools implementing OTTR-support. Extensions to OTTR called bOTTR and tabOTTR enable support for all support structured input such as CSV files, Excel files, and SQL for OTTR, but process mappings that read such data atomically [12]. Van Assche et al. find that many mapping tools have support for data transformations. These are functions that are applied to input values to transform them before they are encoded in triples [11]. The authors cite the need to extract a year from a date when the target schema requires a birth year, while the input contains the birth date. There are problems with the atomic execution of mappings both in the exploratory phase and when making mappings production ready. In the exploratory phase, it is useful to observe intermediary results. For instance, when importing and mapping data from SQL, one will typically perform multiple iterations of adjusting the query and the mapping to get the correct end result. Validating the results of the query is very useful in this process. If there are data transformations involved, the utility of intermediary results in troubleshooting the mapping increases. When creating production-ready code, mapping engineers need maintainable mapping code with logging, data validation as well as tests. Atomic executions however, make it difficult to accomplish these tasks within the mapping process. For instance, it is hard to add validations for the results of SQL queries, to log intermediary statistics such as the number of SQL results and time taken, and to add unit tests. Either the scope of the mapping tool must increase to cover such functionality, or users must move the boundaries of atomic execution and rely on the file-based import of preprocessed CSVs. Alternatively, the mapping language could be extended with a testing facility, but this would greatly increase the scope and complexity of the mapping language.

The review of mapping tools by Van Assche et al. [11] found at the time that only Morph-CSV was based on Python, but it has CLI-based integration. Some of the tools described by Van Assche et al. do have library-based integration, but these are written in Java and have no Python library with bindings. The more recent Morph-KGC allows library-based Python interaction, but is bound by RML requirements to do integrations as specified in the mapping files. With existing tooling then, the way to pass a DataFrame to a mapping tool is to edit the mapping document to point to a CSV-file in the file system and write the DataFrame to CSV. In case the mapping tool is CLI-based, one executes the mapping through the Python subprocess module [62], which permits calling local executables. With SPARQL Anything, the mapping is executed by sending a request to a HTTP-endpoint. In the case of Morph-KGC, one has to call the materialize-method that points to a configuration file. Morph-KGC reads this configuration file, which then points to the RML mapping file which then points to the CSV file. RML mapping files are in any case stored on disk, and one has to leave the notebook environment to edit them. CSV files are not typed, so even though one has taken care to properly parse the DataFrame-columns, they must be parsed again inside the mapping file - this time with another set of parsing functions that the data engineer must look up.

In this section, we have thus far discussed mapping backends, but there are also mapping editors that permit interactivity, but none meet all of our requirements. An open-source R2RML mapping editor described by Sengupta et al. [63] allowed users to create a mapping using a wizard which provides feedback at each step. The wizard allows the user to view the results of SQL queries as well as an interactive view of the generated triples as the mapping is being created. The work was published approximately ten years ago at the time of writing, and we were unable to find the source code for the editor. More recently, Heyvaert et al. [64] created a mapping rule visualization tool that was able to provide better visibility into the input data sets used by RML mappings, as well as the resulting triples. In contrast to wizard-based editors, Heyvaert et. al supports non-linear workflows where users have both source definitions, mappings, and produced triples available at the same time. There was however no support for performing and displaying manipulations on the input data set, as the editor executes the mapping using an end-to-end RML processor. Additionally, the tool has no support for inspecting the results by querying. Ontopic Studio is a commercial offering that also provides an interactive GUI-based way of exploring the data source and the mapped knowledge graph with SPARQL queries [65]. Their tool is however specialized for the setting of virtual knowledge graphs, not materializations.

V. SOLUTION APPROACH AND IMPLEMENTATION

Our solution approach is to perform stateful mapping, where what we call a mapping state is updated by calling a simple

API. In this section, we first describe our chosen mapping language OTTR in Section V-A. We describe the API through a simple example in Section V-B. The template expansion process is described in detail in Section V-C, we discuss implementation details in Section V-D and we compare our solution with Morph-KGC in Section V-E.

A. OTTR

Multiple mapping languages could in principle be supported by the API, but we have chosen the OTTR mapping language. We describe how this choice supports our requirements below. Whereas RML is designed for tight coupling with data sources in order to support SPARQL queries to be executed over source relational databases, stOTTR is designed with loose coupling through an API in mind. This design feature makes it much better suited for composition in a data engineering workflow. Additionally, OTTR supports our need for reusable templates in industrial mapping scenarios (cf. [12], [32], [33]). We use the terse syntax for OTTR, which is called stOTTR [66]. By requiring the use of OTTR, our implementation will meet R4. OTTR does not support interdependencies such as joins between mapping rules. This lacking feature is in fact an advantage over RML in our context. It is likely that data engineers with experience in DataFrame-based tools prefer to perform joins using these DataFrames where they can be worked with and validated interactively instead of inside an atomically executed mapping language where join results can only be inspected by inspecting the generated knowledge graph. As such, lacking this feature helps us support R1 of interactive mapping.

Through industrial experience, we have found alignment problems that are much harder than can be handled with joins defined in RML. In industry, there are typically distinct systems managing operational data flows and asset structure. The naming schemes for individual streams of industrial data (tags) could be based on loosely followed company internal standards and be constrained in their amount of characters allowed by legacy communication protocols such as MODBUS [67], whereas an asset management system has a different way of identifying the data stream. Connecting data from asset management to operational data thus requires advanced string-matching tools or even lookup tables created from analog documentation. Pushing interdependent computations outside of the mapping framework, we simplify our tooling and create a uniform workflow for industrial settings. Second, the lack of interdependencies makes the OTTR mapping problem trivially parallelizable, which makes it much easier to improve the performance of the mapping tool, allowing us to meet R6 with greater ease.

B. API

To initialize a mapping state, a constructor takes named mapping rules as input, which crucially are not connected to input data. The API consists of an operation for applying a mapping rule to a DataFrame of inputs in order to map

triples, an operation for querying the triples in the mapping state using SPARQL, an operation for adding triples with a SPARQL construct query, and data output methods. Fig. 1 shows the operations for adding triples, querying them, and enriching the knowledge base.

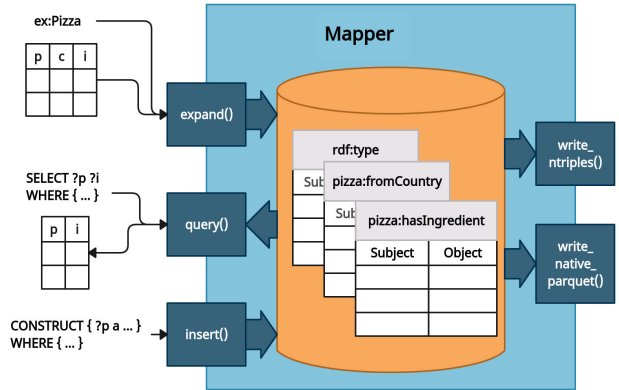


FIGURE 1. The maplib API.

The constructor is used to create a mapping state, and takes a list of stOTTR documents as arguments. A stOTTR document is a string that typically contains prefixes and stOTTR templates. An example abbreviated stOTTR-document together with an instantiation of a Mapping object in Python is given in Listing 1. We have based this example on the example templates for mapping knowledge about Pizzas in Skjæveland et al. [12], which the authors use to introduce OTTR and the stOTTR-syntax. We note that the DataType annotation `xsd:AnyURI` on the `?c` parameter is necessary since country occurs only in the object position and may be a string literal or an IRI. The same goes for the ingredient parameter `?is`, except this datatype is nested in a list. In OTTR, template rule applications are called template expansions, which is why the API method for rule application is called `expand`. The `expand` method takes as input a template identifier (IRI or prefixed) together with an Apache Arrow-based DataFrame with corresponding columns as input. Note that Apache Arrow is only a specification of how columnar data is represented in-memory, and that multiple implementations exist.

```
doc = """
...
ex:Pizza[?p, xsd:AnyURI ?c,
    List<xsd:AnyURI> ?is] :: {
  ottr:Triple(?p, a, pizza:Pizza),
  ottr:Triple(?p, pizza:fromCountry, ?c),
  cross | ottr:Triple(?p, pizza:hasIngredient, ++?is)
}.
...
"""

m = Mapping([doc])
```

Listing 1. Example instantiation of mapping-object using an abbreviated template from [12].

Apache Arrow specifies a set of datatypes, including list-types. We expect the input DataFrame to use these types, and not to perform string-encoding of input data. If we want to expand the template with heterogeneous types in one of the arguments, we must do separate calls to `expand`. From the datatype of a column, an implementation should infer the OTTR datatype (and consequently the RDF datatype) of the argument if it is not specified in the template signature. Inconsistent datatypes should produce an error. This type-inference also extends to Apache Arrow list-types. In sum, this API design allows us to support R2 of interoperating with data engineering tools in a comprehensive way.

We build a very simple dataset below in Listing 3, and expand the `ex:Pizza` template. We first define some prefixes in Listing 2.

```
ex = "https://.../maplib/example#"
co = "https://.../maplib/countries#"
pi = "https://.../maplib/pizza#"
ing = "https://.../maplib/pizza/ingredients#"
```

Listing 2. Example dataset and template expansion.

```
df = pl.DataFrame({
    "p": [pi + "Hawaiian", pi + "Grandiosa"],
    "c": [co + "CAN", co + "NOR"],
    "is": [[ing + "Pineapple", ing + "Ham"],
          [ing + "Pepper", ing + "Meat"]]
})
print(df)
m.expand("ex:Pizza", df)
```

Listing 3. Example dataset and template expansion.

The output of printing the DataFrame looks approximately like Table 1. Now, we can inspect the model using a SPARQL query.

TABLE 1. Abbreviated printout of the DataFrame `df`.

p (str)	c (str)	is (List[str])
"https://...#Hawaiian"	"https://...#CAN"	["..#Pineapple", "..#Ham"]
"https://...#Grandiosa"	"https://...#NOR"	["..#Pepper", "..#Meat"]

```
qdf = m.query("""
PREFIX pizza:<https://.../maplib/pizza#>
SELECT ?p ?i WHERE {
?p a pizza:Pizza .
?p pizza:hasIngredient ?i .
}
""")
print(qdf)
```

Listing 4. Querying the mapping state.

The SPARQL query produces results given in Table 2.

Next the model may be enriched using the `insert` method. First, we define a construct-query that we want to enrich the model with, and execute it to see that it produces the expected results.

TABLE 2. Results of the printout of the inspection query `qdf`.

p (str, IRI)	i (str, IRI)
"https://...#Hawaiian"	"https://...#Pineapple"
"https://...#Hawaiian"	"https://...#Ham"
"https://...#Grandiosa"	"https://...#Pepper"
"https://...#Grandiosa"	"https://...#Meat"

```
hpizzas = """
PREFIX pizza:<https://.../maplib/pizza#>
PREFIX ing:<https://.../maplib/pizza/ingredients#>
CONSTRUCT { ?p a pizza:UnorthodoxPizza }
WHERE {
    ?p a pizza:Pizza .
    ?p pizza:hasIngredient ing:Pineapple .
}"""
res = m.query(hpizzas)
print(res[0])
```

Listing 5. Querying the mapping state.

TABLE 3. Abbreviated printout the construct query results `res`.

subject (str, IRI)	verb (str, IRI)	object (str, IRI)
"https://...#Hawaiian"	"..#type"	"https://...#UnorthodoxPizza"

This query produces a list of DataFrames as a result with length one. The contained DataFrame is shown in Table 3.

Since the results are as expected, we enrich the model and check that it is updated in Listing 6. The results of this query are shown in Table 4. Note that the `insert` method accepts construct-queries in addition to insert-queries, as this allows us to use the same construct-query to verify and to insert new triples without modifying the string.

```
m.insert(hpizzas)
m.query("""
PREFIX pizza:<https://.../pizza#>

SELECT ?p WHERE {
?p a pizza:HeterodoxPizza
}
""")
```

Listing 6. Querying the mapping state.

TABLE 4. Abbreviated printout the construct query results `res`.

p (str)
"https://...#Hawaiian"

Finally, we may export the mapping state either to an N-Triples-file or to a collection of Parquet-files using the `write_ntriples` or `write_native_parquet` methods respectively.

C. TEMPLATE EXPANSION

This section looks closely at the template expansion, and how it works in our solution approach. OTTR template expansion can be implemented using a simple recursive function on the

instance or row level, which we will discuss as a baseline algorithm, before we look at the column-based algorithm being used. Listing 7 shows this baseline recursive function written in Python notation. We allow `ottr:Triple` to refer to the corresponding Template object, even though this is not proper Python.

```
def expand_row(template:Template, args:List[str])
    -> List[Triple]:
    if template == ottr:Triple :
        return [Triple(args[0], args[1], args[2])]
    else:
        triples = []
        for pattern in template.patterns:
            new_args = []
            for a in pattern.arguments:
                if is_param(a):
                    a = replace(a, template, args)
                    new_args.append(a)
            triples += expand_row(pattern, new_args)
        return triples
```

Listing 7. Instance-level recursive procedure for expanding OTTR templates.

`expand_row` takes a template as the first argument. A template consists of a signature and a body with a list of patterns [31]. In Listing 1 the signature was between the `[` and `]`-brackets, while the lines with `ottr:Triple(...)` were the patterns. The base condition of this recursive function is when we reach an `ottr:Triple`-template. In this case, we can create the result. Otherwise, we perform recursive calls to `expand_row` after creating a new list of arguments, replacing parameters with their arguments.

We consider a call to `expand_row` with the `ex:Pizza`-template from Listing 1, without considering the list parameter `?is` as it is not covered by Listing 7. The call is shown in Listing 8. We will allow `ex:Pizza` to refer to a `Template` object for simplicity, even though this is not proper Python syntax, as we did with `ottr:Triple` previously.

```
expand_row(ex:Pizza,
           ["https://..#Hawaiian", "https://..#CAN"])
```

Listing 8. An example call to the baseline row-based expansion function.

We iterate through the patterns in our template, (excluding the `CROSS` for simplicity), producing new lists of arguments where parameter variables are replaced by their arguments. This produces new calls to `expand_row`, which are shown in Listing 9.

```
expand_row(ottr:Triple, ["https://..#Hawaiian",
                        "http://..#type", "https://..#Pizza"])
expand_row(ottr:Triple, ["https://..#Hawaiian",
                        "https://..#fromCountry", "https://..#CAN"])
```

Listing 9. New calls to `expand_row` in our example.

The end result is a list of triples as shown in Listing 10. However, in our case, we are expanding templates in a column-based fashion. Although we follow the semantics

```
[Triple("https://..#Hawaiian",
        "http://..#type", "https://..#Pizza"),
 Triple("https://..#Hawaiian",
        "https://..#fromCountry", "https://..#CAN")]
```

Listing 10. Result from calling `expand_rows`.

of OTTR template expansion, there are important differences. The process of expanding OTTR templates in a column-oriented way is presented using Python syntax in Listing 11.

```
class Result:
    df: DataFrame
    const_args: Dict[str, Constant]
    is_unique: bool

def expand_col(t:Template,
              df:DataFrame,
              const_args: Dict[str, Constant],
              unique: Set[str])
    -> List[Result]:
    if t == ottr:Triple :
        is_u = not unique.empty()
        r = Result(df, const_args, is_u)
        rs = normalize_results(r)
        return rs

    prep_pat_args = lambda x:
        prep_args(x, df, const_args, unique)

    args = par_map(prepare_pat_args, t.patterns)
    results = par_map(expand_col, args)
    return flatten(results)
```

Listing 11. Simplified algorithm for recursive expansion of OTTR templates in Python.

The `expand_col` method takes four arguments, the template to expand, the `DataFrame` and `const_args`-dict that together contain exactly the template parameters, and set of column names that are unique. Initially, the `const_args`-dict is empty. The function produces a list of `Result`-objects, which contains the resulting triples. These have a normalized form of having a `DataFrame` with subject and object columns, together with a constant for the verb in the dict `const_args`. The `is_unique` field tells us if the `DataFrame` rows are unique. After we have run the function, we store the results directly in the triplestore. The storage operation (not shown) makes note of whether the stored `DataFrame` has only unique values or not, but does not check whether it is actually the case, as this is an expensive operation.

As in `expand_row`, the base case is when the template is `ottr:Triple`. The `normalize_results`-function (implementation not shown) takes a result and produces a list of normalized results. Normalization is done in order to simplify the implementation of the triplestore. We normalize each result into one or more `DataFrame`s with a subject and object column - converting constants to `DataFrame` columns. In case the verb is a column (not a constant) we convert the result into as many results as there are unique verb IRIs by partitioning the result `DataFrame` on the verb-column.

In the future, we may allow heterogeneous representations of verbs in the database with constant subjects or objects. However, this causes a lot of complexity in query processing logic which we have chosen to avoid for the time being. After results are normalized they can be stored directly in the triplestore by updating a map according to the verb IRI and object-datatype of the result to be stored.

If the base condition has not been reached, we create as many DataFrames as there are new templates to instantiate in the body of the template using the `prep_pat_args`-function, selecting and renaming the appropriate columns. We create new dictionaries of renamed constant arguments and keep track of which of the new columns are unique. The `par_map` function denotes that this operation is done in parallel and collected in a list. This parallelism comes in addition to the parallelism inherent in Polars operations. Next, we execute the `expand_col` function (also in parallel) for the prepared tuples of arguments. Finally, we must flatten the resulting list of lists of results into a list of results, which we return.

We are now ready to consider an example execution of the `expand_col`-method in Listing 12. The `df`-argument is the DataFrame from Table 1.

```
expand_col(ex:Pizza, df, {}, {"p"})
```

Listing 12. First call to `expand_col` in our example.

Continuing our example, we prepare the argument-lists shown in Listing 13.

```
[[ottr:Triple, df_1, {"verb": "http://..#type",
  "object": "https://..#Pizza"}, {"subject"}],
 [ottr:Triple, df_2, {"verb":
  "https://..#fromCountry"}, {"subject"}],
 [ottr:Triple, df_3,
  {"verb": "https://..#hasIngredient"}, {}]]
```

Listing 13. List of list of arguments created for the `ex:Pizza`-template.

TABLE 5. Abbreviated printout of the DataFrame `df_1`.

subject (str)
"https://..#Hawaiian"
"https://..#Grandiosa"

TABLE 6. Abbreviated printout of the DataFrame `df_2`.

subject (str)	object (str)
"https://..#Hawaiian"	"https://..#CAN"
"https://..#Grandiosa"	"https://..#NOR"

We display the DataFrames `df_1`, `df_2` and `df_3` in Tables 5, 6 and 7 respectively. These are created in parallel, and data are only copied as necessary. The DataFrame `df_3` is the result of “exploding” the `is`-column of `df`, which is a built-in function in Polars, duplicating other

TABLE 7. Abbreviated printout of the DataFrame `df_3`.

subject (str)	object (str)
"https://..#Hawaiian"	"https://..#Pineapple"
"https://..#Hawaiian"	"https://..#Ham"
"https://..#Grandiosa"	"https://..#Pepper"
"https://..#Grandiosa"	"https://..#Meat"

columns as necessary. This means that the `subject`-column (renamed from `p`) no longer can be guaranteed to be unique, leading to an empty `unique`-set. Finally, we call the `expand_col`-function in parallel with the arguments above. We reach the base case of the `expand_col`-method, and the object-column is added to `df_1` in the normalization process (`normalize_results`). The normalization process does not change the other resulting DataFrames. These resulting DataFrames are ready to store and query once we ensure that the DataFrame associated with `pizza:hasIngredient` is deduplicated.

D. IMPLEMENTATION DETAILS

We implement the solution in Rust, but create Python bindings where data are exchanged using Apache Arrow Interprocess Communication (IPC), meaning data are not copied [35]. We base our SPARQL engine on an engine used in our previous work with Apache Arrow-based SPARQL-processing [16]. It relies on the Polars-library for representing triples and processing queries, and on the `spargebra` [68] and `oxrdf` [69] libraries for parsing SPARQL expressions and representing RDF data. The solution is available on GitHub under the Apache 2.0 License.¹

The real column-based expansion procedure keeps track of the data types of the arguments. We store the triples created by mapping as a verb and data type-indexed collection of Polars DataFrames. Associated with each verb-IRI is a map from data types to Polars DataFrames containing the columns subject and object of the predicate, where the object has the given data type. Column-based triple storage has been successfully adopted by previous approaches to analytic / read-only SPARQL querying [70], [71].

Before SPARQL processing can begin, we deduplicate the triplestore. We will however skip deduplication for a verb if it is already deduplicated or the user has (indirectly) indicated that the triples associated with the verb are unique. When processing SPARQL queries, we create a lazy expression corresponding to the SPARQL algebra-expression [22] of the query as provided by the `spargebra`-library [68]. The semantics of SPARQL-algebra can be mapped mostly directly to Polars operations on DataFrames. We thus rely on optimized parallel execution by Polars for actual processing. We follow the best practice advice of Polars and use a global string cache managed by Polars for Strings (e.g. IRIs), and use instead categorical integer values in place of them to perform joins.

¹<https://github.com/magbak/maplib>

Our SPARQL implementation does not yet support solution mappings where a single variable has multiple data types. Such support adds a lot of complexity to query processing as we need to keep track of whether we are using native types or whether we are using the native- or the string encoding of a variable. This support will be added in a future release.

E. COMPARISON WITH EXISTING APPROACHES

The column-oriented graph construction algorithm described above differs from Morph-KGC in important ways. Morph-KGC also employs a column-oriented approach to mapping, and applying RML mappings to DataFrames backed by Pandas. However, data is loaded into a SPARQL database by writing to an in-memory ntriples-file, which is then loaded in bulk.² Before the ntriples-file is created, data are also deduplicated. We avoid these expensive operations by re-using the target representation for mapping as the underlying format for the SPARQL engine and by allowing the user to specify uniqueness. In many cases, we copy very little to no data during template expansions, and merely change how we relate to it. This is for instance the case for the c-column in Listing 1. When data is copied, the sequential nature of the columnar format likely means copying is very fast. In order to support RML, Morph-KGC also must support joins as part of the mapping, but there is no such support for joins in OTTR, which we implement. We return to this difference and what it implies in our discussion in Section VI-C.

When SPARQL Anything is used to materialize the knowledge graph, Apache Jena is indirectly used to query the source data set files and update the graph. From the highly abstract and configurable Apache Jena source code it is much harder to tell what this process actually involves, and an expert or detailed study of the Jena code base is likely required to determine that with certainty.

VI. EVALUATION

In this section, we evaluate the degree to which our solution meets R6 of high performance for large datasets. We evaluate this requirement by comparing our solution to two state-of-the-art approaches on a demanding benchmark:

- The Morph-KGC approach of Arenas-Guerrero et al. [9].
- The SPARQL Anything-approach of Asprino et al. [17].

In Section VI-A we describe how we set up and ran the benchmark. Our results are presented in Section VI-B and discussed in Section VI-C. Code and instructions for running the benchmark and our raw results are available on GitHub.³

A. METHODS

We compare performance on the General Transit Feed Specification (GTFS)⁴ Madrid benchmark described by [18],

²https://github.com/morph-kgc/morph-kgc/blob/751bf7/src/morph_kgc/_init_.py

³https://github.com/magbak/maplib_paper_benchmarks

⁴<https://gtfs.org/>

which contains a generator of scalable datasets based on openly accessible data from the public transport system in Madrid. The GTFS is a standardized format that public transport organizations can use to publish data, initially created at Google [72]. The GTFS Madrid benchmark contains both a RML mapping from CSV files and 18 SPARQL queries over the mapped knowledge graph. We use the same scaling factors (5, 10, 50, 100, 500) as are used in the benchmark paper [18].

We compare maplib with Morph-KGC [9] and SPARQL Anything [17]. In order to benchmark maplib, we created a stOTTR mapping equivalent to the RML mapping used by Chaves et al. [18]. We verify that the mapped models are identical up to small differences in datatypes due to differing defaults in maplib and Morph-KGC (e.g. `xsd:long` instead of `xsd:integer`), and crucially, that queries produce the same rows. We eliminated query 10 from consideration, as the generated data and the query do not agree on data types (comparing `xsd:string` with `xsd:duration`).

SPARQL Anything has a GTFS mapping in a code repository [73] associated with a recent GTFS benchmark [74], which is specified as a single SPARQL query annotated with a service that refers to the GTFS CSV files. However, running the joint query did not terminate in a reasonable time-frame (30+ minutes and still running). To achieve comparable performance, we split the joint materialization query into a materialization query for each GTFS CSV file. SPARQL Anything exists in a stateless- and as a server-based variant. The server-based variant is more appropriate for our setting. Running SPARQL Anything Server, we first run the ten materialization queries and then the GTFS queries using the SPARQLWrapper-library in Python.

Our evaluation consisted of materializing the knowledge graph and executing the GTFS queries in 10 separate executions. We set a timeout of ten minutes for knowledge graph materialization. We recorded wall times for materialization and for executing each query, as interactive creation of the mapping often requires (re-)execution of mapping and queries. We set a timeout of two minutes for query execution. The evaluation was run on a high end laptop from 2017 with an Intel i7-7600U CPU @ 2.80GHz, 16GB of ram, and a high-performance SSD. We used the 0.3.12 release of maplib which uses Polars 0.26.1, the 2.3.1 release of Morph-KGC, and the 0.8.1 version of SPARQL Anything Server. Morph-KGC supports both RDFLib and Oxigraph triplestore backends. Morph-KGC 2.3.1 uses RDFLib version 6.2.0 and pyoxigraph 0.3.10. SPARQL Anything 0.8.1 uses Apache Jena Fuseki 4.2.0.

Morph-KGC requires the RML-file to be located in the file system, and the parameters to the materialization function to be specified in a.ini-file. Thus, Morph-KGC will read two extra files, but these are very small compared to the GTFS data sets, and we do not expect this difference to be reflected in our results. Similarly, SPARQL Anything reads ten very small files with the SPARQL queries necessary to materialize the GTFS graph.

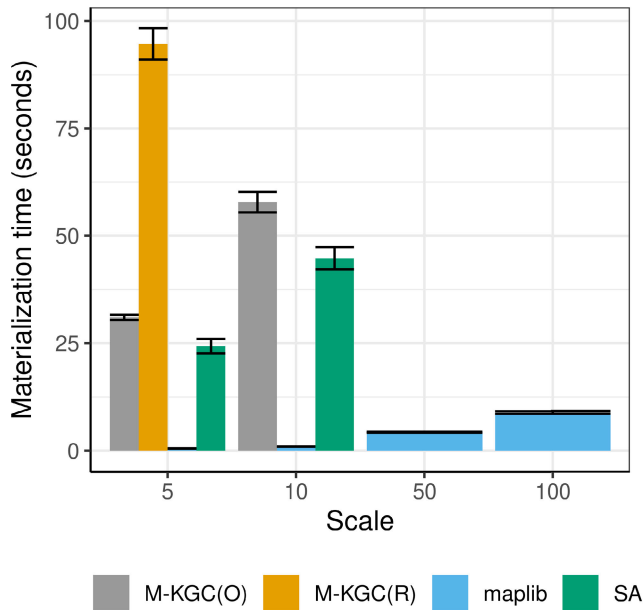


FIGURE 2. Mean materialization times, error bars are one standard deviation away from the mean.

B. RESULTS

The results from executing the materialization stage are given in Fig. 2, and the mean materialization times are given in Table 8. We denote Morph-KGC with a RDFLib materialization by “M-KGC(R)” and Morph-KGC with an Oxigraph materialization by “M-KGC(O)”. SPARQL Anything is denoted by “SA”. Following the convention of Chaves-Fraga et al. [18], “TO” in a table cell indicates that the materialization took longer than 10 minutes, and use M to denote the case where materialization ran out of memory. X indicates that a scale was skipped due to a smaller scale having timed out. Morph-KGC also reported a mapping-rule processing time of four seconds, relating to the partitioning approach described in Arenas-Guerrero et al. [9].

Mean query execution times (seconds) for each query, scale, and solution are presented in Table 9. The variability of execution times is small, and we omit the standard deviations from the table for readability. In the table, N indicates that the query is not supported by the implementation in question. This is only the case for maplib and query 15. X indicates that the query was not run due to missing materialization. M indicates that query processing ended in an error of the kind one gets from Rust when it has run out of memory. TO indicates that the query used more than the allotted two minutes to complete.

C. DISCUSSION

The results of our evaluation show a 47×, 60×, and 182× performance improvement in maplib over SPARQL Anything, Morph-KGC with Oxigraph, and Morph-KGC with RDFLib respectively. Moreover, our solution is able to materialize larger data sets than both Morph-KGC and SPARQL

TABLE 8. Table of mean materialization times (seconds) with standard deviations in parentheses.

Scale	M-KGC(O)	M-KGC(R)	SA	maplib
5	31.0 (0.6)	94.7 (3.64)	24.3 (1.67)	0.52 (0.07)
10	57.8 (2.39)	TO	44.8 (2.58)	0.96 (0.05)
50	TO	TO	TO	4.34 (0.1)
100	X	X	X	8.9 (0.31)
500	X	X	X	M

Anything, effectively allowing us interactively work with larger data sets. Maplib materialization time is less than ten seconds even for the largest successfully completed materialization.

Query processing times are on the whole in favor of SPARQL Anything and Morph-KGC (with Oxigraph) over maplib. Oxigraph and SPARQL Anything are able to do considerably better than maplib on many queries in terms of time and memory (Q3, Q5, Q6, Q7, Q8, Q11, Q12, Q17 and Q18). Morph-KGC with RDFLib times out on a lot of queries, but achieves comparable performance to the other query engines on many of the queries where it does not time out. Maplib is however able to do considerably better than Oxigraph and SPARQL Anything on queries Q1, Q2, Q9, Q13, Q14 and Q16. RDFLib is able to do considerably better than Oxigraph and maplib on Q11 and Q13.

In an interactive knowledge graph construction session, the engineer likely re-runs both knowledge graph construction and query evaluation(s) as a whole. The purpose of the query is precisely to inspect the effect of the mapping change. Mapping times and query execution times should be considered in conjunction. Considering materialization times and query times jointly, it is clear that the materialization time differences dominate the query performance differences in all but one query (Q13), leaving maplib with a performance advantage for the joint problem of materialization and querying. Moreover, the materialization process in maplib is much more scaleable than those of SPARQL Anything and Morph-KGC.

Both RML and SPARQL Anything specify the source CSV files as part of the mapping definition files (Turtle and SPARQL respectively). stOTTR does not describe input data files, and we give paths to input files in the Python script. To run Morph-KGC and SPARQL Anything on different data sets, we either had to adjust the mapping files or we had to move a different data set into the data directory. We found that this process impeded our productivity. This setup is also likely to complicate development environments where the same mapping is tested with different inputs. To test the RML or SPARQL Anything mapping with different data sets, for instance against a data set that contains an error we want to handle, we must copy the mapping files into an adjacent directory or change the source mapping-files to point to different CSV source files.

While the partitioning approach of Arenas-Guerrero et al. [9] is impressive, it comes at a performance cost (about

TABLE 9. Table of query processing times in seconds.

Query	Scale	M-KGC(O)	M-KGC(R)	SA	maplib
Q1	5	6.65	30.65	7.95	3.06
Q1	10	21.10	X	15.81	6.75
Q1	50	X	X	X	34.29
Q1	100	X	X	X	71.63
Q2	5	2.87	1.63	0.58	1.59
Q2	10	10.64	X	0.95	3.52
Q2	50	X	X	X	17.96
Q2	100	X	X	X	36.78
Q3	5	0.18	1.78	0.19	2.49
Q3	10	0.57	X	0.27	5.53
Q3	50	X	X	X	28.11
Q3	100	X	X	X	57.57
Q4	5	0.02	1.45	0.03	0.62
Q4	10	0.05	X	0.03	1.26
Q4	50	X	X	X	5.77
Q4	100	X	X	X	11.76
Q5	5	0.01	0.08	0.02	0.08
Q5	10	0.02	X	0.03	0.15
Q5	50	X	X	X	0.67
Q5	100	X	X	X	1.31
Q6	5	0.00	0.01	0.01	0.07
Q6	10	0.01	X	0.01	0.13
Q6	50	X	X	X	0.61
Q6	100	X	X	X	1.14
Q7	5	0.74	TO	0.05	M
Q7	10	5.12	X	0.05	M
Q8	5	3.66	TO	7.26	25.14
Q8	10	10.95	X	15.01	M
Q9	5	28.72	TO	35.73	9.05
Q9	10	102.59	X	M	19.87
Q9	50	X	X	X	M
Q11	5	0.35	0.14	0.17	2.23
Q11	10	1.12	X	0.23	4.62
Q11	50	X	X	X	22.84
Q11	100	X	X	X	47.85
Q12	5	0.41	TO	0.15	11.26
Q12	10	1.56	X	0.17	M
Q13	5	TO	0.59	0.17	98.89
Q13	10	X	X	0.22	M
Q14	5	0.90	TO	1.44	0.58
Q14	10	2.92	X	2.72	1.19
Q14	50	X	X	X	5.94
Q14	100	X	X	X	11.77
Q15	5	1.73	3.52	0.60	N
Q15	10	3.78	X	1.25	N
Q16	5	0.88	38.16	1.24	0.18
Q16	10	2.04	X	0.26	0.34
Q16	50	X	X	X	1.30
Q16	100	X	X	X	2.44
Q17	5	0.29	TO	0.19	8.38
Q17	10	0.97	X	0.25	34.88
Q17	50	X	X	X	M
Q18	5	0.02	0.06	0.04	0.08
Q18	10	0.06	X	0.06	0.15
Q18	50	X	X	X	0.69
Q18	100	X	X	X	1.37

4 seconds in the benchmark, independent of scale). In a one-way data engineering setting that materializes triples,

it can be avoided altogether. In the GTFS-Madrid benchmark for instance, the `STOPS.csv`-file contains the `stop_id`-column. The `STOP_TIMES.csv` also contains this column, with a foreign key relationship to `stop_id`. A static prefix is used by the RML-mapping to construct the IRI for a stop based on data in `STOPS.csv`, and `stop_id` in `STOP_TIMES.csv` is used to reference this constructed IRI. If we can guarantee the foreign key constraint between these files, and have sufficient information to construct the IRI in both of them, the IRIs can be computed independently for both files without the need for a join. This is the approach we have chosen in our benchmark implementation for maplib and SPARQL Anything. IRI construction is string concatenation which scales linearly and is easily parallelizable. Joins however, do not scale linearly. A bad query execution plan for these joins may have been the cause of the long-running materialization of SPARQL Anything that we had to break up (cf. Section VI-A).

If on the other hand, the table containing the foreign key (`STOP_TIMES.csv`) does not have sufficient information to construct the IRI of the referenced table (`STOPS.csv`), the join is mandatory. In this case, we can join in the necessary data before we start mapping, and construct the mapping such that it contains no join. This strategy comes at the expense of having to create the IRI twice, but simplifies and improves the parallelism in the mapping process. Future research into RML parallelization should compare the above strategy of getting rid of dependencies early with the approach of Arenas-Guerrero et al. [9].

With an interactive mapping process in Jupyter, users should be able to compute this amended data set as part of their general data engineering workflow. If the join is expensive, they can place it at the very start of the workflow or store the amended dataset and start the workflow by importing it, effectively eliminating the need to recompute the join in further mapping iterations.

The choice of Polars over Pandas likely plays a role in the materialization performance difference between Morph-KGC and maplib, as this library is known to be faster [15]. To further understand the difference, we added time measurements to the process Morph-KGC uses to create the ntriples-string and load this string into Oxigraph (cf. Section V-E). In a single sample run for scale 10 that took 48.91 seconds for oxigraph, 0.86 seconds were spent creating the ntriples string, and a further 22.75 seconds were spent bulk-loading the ntriples-string into Oxigraph. The extra work done by Morph-KGC described above on e.g. joins, as well as the collection of all produced triples in a set described in Section V-E likely also play a role. However, there is also a role to be played by Polars. Comparing csv-load times, Polars takes approximately 0.06 seconds to load the CSVs at scale 10, and Pandas takes approximately 0.3 seconds. Similar differences likely exist for operations that build the IRIs in the mapping and may well add up to several seconds at scale 10. However, it appears likely that the fact that maplib does less work is responsible for the

majority of the difference between maplib and Morph-KGC in materialization times, and not that the work done by both solutions takes less time in maplib.

VII. CONCLUSION AND FUTURE WORK

In this work we have described maplib. Maplib allows users to perform interactive, literal mapping and enrichment of knowledge graphs through SPARQL, allowing us to meet our requirements of supporting interactive mapping in a script-based language with SPARQL based inspection and post-processing (R1, R3 and R5). By using OTTR instead of RML or Façade-X as the mapping language, we allow greater interactivity and greatly simplify the integration with DataFrames compared to the state-of-the-art Morph-KGC and SPARQL Anything libraries, allowing us to meet R2. Using OTTR also allows for improved and less complicated parallelism in the mapping process compared to Morph-KGC and SPARQL Anything. Compared to RML, the cost of this choice is that our mappings cannot be used to rewrite SPARQL queries into SQL queries over a source database. Additionally, OTTR has excellent template support, allowing us to support R4 for industrial use cases.

Using the Polars-library, we are able to use the same representation to store mapped triples and query them, with minimal data transformation and copying. We show that our solution is able to outperform the state-of-the-art Morph-KGC and SPARQL Anything libraries by a factor of 47x in the materialization step and in terms of scalability. Query execution is less effective with maplib, but these differences are dominated by the relative difference in materialization time in all but one query, allowing maplib to exhibit much shorter feedback loops in exploratory knowledge graph construction than Morph-KGC and SPARQL Anything. These performance increases allow us to meet R6 of high performance for large datasets. The performance improvements over Morph-KGC and SPARQL Anything are not the results of any clever algorithms created by the author, but a result of simplifying the mapping process and providing greater interoperability with Apache Arrow-based DataFrames, meaning we have to do less data transformation work and have greater access to high performance tooling such as Polars.

Our experience indicates that RML and Façade-X-based mapping is currently ill-suited for integration in interactive, literal Python-based workflows commonly used by Data Engineers. Using it in this way incurs toil on the part of the user, limits the interactivity and literal feedback from the mapping process and incurs a performance penalty. Additionally, the design of RML and Façade-X makes it harder to make the workflow production ready once the exploration stage is completed. The OTTR approach is better suited for this purpose by virtue of the functional, API-based design.

Maplib is currently limited to a single machine, but we are looking into a distributed, scalable implementation. The same API can be implemented in a distributed setting, reusing parts of the implementation. The mapping step is embarrassingly parallel and can be scaled easily. We have shown the benefit

of Apache Arrow-based integrations over an approach that requires data transformation. This benefit can be extended to a distributed setting as Polars DataFrames produced by maplib can be immediately exported to cloud object storage as Parquet in a folder structure partitioned by predicate IRI and data type. Exporting from Arrow to Parquet is very fast. This ensures distributed support for the `expand-method`. Next, a data lakehouse such as Dremio can be configured to query the data set using SQL. With this in place, a mapping for a Virtual Knowledge Graph (e.g. Ontop) can be automatically constructed, so that the dataset can be queried using SPARQL, and support for the `query` and `insert` methods can thus be added.

While the support for interactive SPARQL queries means we are able to perform validations of mapping output, the SHACL specification defines a set of validation constraints that make such validations easier to define [75]. SHACL constraints are also a natural match for reusable mapping templates in industry, as they help us ensure that new users of the templates use them correctly, and in accordance with an underlying industrial standard [12]. In future work, we therefore plan on adding SHACL support. Having access to highly expressive and performant Polars primitives likely makes it possible to do so in a performant way.

Similarly, support for reasoning could likely benefit industrial applications. Supporting reasoning could e.g. be useful to create auxiliary structures that further harmonize and standardize the model, and thus improve interoperability. We plan on exploring support for reasoning in future work.

ACKNOWLEDGMENT

The author would like to thank Francisco Martin-Recuerda, Ph.D. at SINTEF Digital in Oslo, Norway, for making them aware of the lack of interactive Python-based RDF-mapping tools in the literature, also would like to thank Martin G. Skjæveland, Ph.D. with the Centre for Scalable Data Access, University of Oslo, for helpful advice on the direction and framing of the work, also would like to thank Dylan Van Assche with the IDLaboratory, Ghent, Belgium, for his useful suggestions on benchmarking, and also would like to thank Professor Ahmet Soylu with Oslo Metropolitan University for constructive feedback on the work.

REFERENCES

- [1] T. Hubauer, S. Lamparter, P. Haase, and D. M. Herzig, "Use cases of the industrial knowledge graph at Siemens," in *Proc. ISWC (P&D/Industry/BlueSky)*, 2018.
- [2] S. R. Bader, I. Grangel-Gonzalez, P. Nanjappa, M.-E. Vidal, and M. Maleshkova, "A knowledge graph for industry 4.0," in *Proc. Semantic Web, 17th Int. Conf. (ESWC)*. Heraklion, Crete, Greece: Springer, Jun. 2020, pp. 465–480.
- [3] *Project Jupyter*. Accessed: Jan. 25, 2023. [Online]. Available: <https://jupyter.org/>
- [4] D. E. Knuth, "Literate programming," *Comput. J.*, vol. 27, no. 2, pp. 97–111, 1984.
- [5] J. M. Perkel, "Why Jupyter is data scientists' computational notebook of choice," *Nature*, vol. 563, no. 7732, pp. 145–147, 2018.
- [6] TIOBE. *TIOBE Index for January 2023*. Accessed on: Jan. 18, 2023. [Online]. Available: <https://www.tiobe.com/tiobe-index/>

- [7] W. McKinney, "Pandas: A foundational Python library for data analysis and statistics," *Python High Perform. Sci. Comput.*, vol. 14, no. 9, pp. 1–9, 2011.
- [8] Pandas. *API Reference—Pandas 1.5.3 Documentation*. Accessed: Jan. 25, 2023. [Online]. Available: <https://pandas.pydata.org/docs/reference/index.html#api>
- [9] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, M. S. Pérez, and O. Corcho, "Morph-KGC: Scalable knowledge graph materialization with mapping partitions," in *Proc. Semantic Web*, 2022, pp. 1–18.
- [10] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. Van de Walle, "RML: A generic language for integrated RDF mappings of heterogeneous data," in *Proc. 7th Workshop Linked Data Web*, 2014.
- [11] D. Van Assche, T. Delva, G. Haesendonck, P. Heyvaert, B. De Meester, and A. Dimou, "Declarative RDF graph generation from heterogeneous (semi-)structured data: A systematic literature review," *J. Web Semantics*, vol. 75, Jan. 2023, Art. no. 100753.
- [12] M. G. Skjæveland, D. P. Lupp, L. H. Karlsen, and J. W. Klüwer, "OTTR: Formal templates for pattern-based ontology engineering," in *Proc. WOP*, 2021, pp. 349–377.
- [13] J. Arenas-Guerrero. *Powerful RDF Knowledge Graph Generation With [R2]RML Mappings*. Accessed: Jan. 25, 2023. [Online]. Available: <https://github.com/morph-kgc/morph-kgc>
- [14] R. Vink. *Polars*. Accessed: Jan. 26, 2023. [Online]. Available: <https://www.pola.rs/>
- [15] *Polars: TPCH Benchmarks*. Accessed: Jan. 26, 2023. [Online]. Available: <https://www.pola.rs/benchmarks.html>
- [16] M. Bakken and A. Soylu. (Dec. 2022). *Chrontext: Portable SPARQL Queries Over Contextualised Time Series Data in Industrial Settings*. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4310978
- [17] L. Asprino, E. Daga, A. Gangemi, and P. Mulholland, "Knowledge graph construction with a façade: A unified method to access heterogeneous data sources on the web," *ACM Trans. Internet Technol.*, vol. 23, no. 1, pp. 1–31, 2022.
- [18] D. Chaves-Fraga, F. Priyatna, A. Cimmino, J. Toledo, E. Ruckhaus, and O. Corcho, "GTFS-Madrid-bench: A benchmark for virtual knowledge graph access in the transport domain," *J. Web Semantics*, vol. 65, Dec. 2020, Art. no. 100596.
- [19] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Sci. Amer.*, vol. 284, no. 5, pp. 34–43, May 2001.
- [20] P. Hitzler, M. Krtzsch, and S. Rudolph, *Foundations of Semantic Web Technologies*, 1st ed. Boca Raton, FL, USA: Chapman & Hall/CRC, 2009.
- [21] L. Ehlringer and W. Wöß, "Towards a definition of knowledge graphs," *SEMANTICS Posters, Demos, Success*, vol. 48, nos. 1–4, p. 2, 2016.
- [22] World Wide Web Consortium. *SPARQL 1.1 Query Language*. Accessed: Jan. 18, 2023. [Online]. Available: <https://www.w3.org/TR/sparql11-query/>
- [23] G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati, and M. Zakharyashev, "Ontology-based data access: A survey," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, Jul. 2018, pp. 1–9.
- [24] G. Xiao, L. Ding, B. Cogrel, and D. Calvanese, "Virtual knowledge graphs: An overview of systems and use cases," *Data Intell.*, vol. 1, no. 3, pp. 201–223, 2019.
- [25] S. Das, S. Sundara, and R. Cyganiak. *R2RML: RDB to RDF Mapping Language*. Accessed: Jan. 18, 2023. [Online]. Available: <https://www.w3.org/TR/r2rml/>
- [26] S. Auer, L. Feigenbaum, D. Miranker, A. Fogarolli, and J. Sequeda. (Jun. 2010). *Use Cases and Requirements for Mapping Relational Databases to RDF*. Accessed: Jan. 18, 2023. [Online]. Available: <https://www.w3.org/TR/rdb2rdf-ucr>
- [27] E. Iglesias, S. Jozashoori, D. Chaves-Fraga, D. Collarana, and M.-E. Vidal, "SDM-RDFizer: An RML interpreter for the efficient creation of RDF knowledge graphs," in *Proc. 29th ACM Int. Conf. Inf. Knowl. Manage.* New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 3039–3046.
- [28] C. Debruyne and D. O'Sullivan, "R2RML-F: Towards sharing and executing domain logic in R2RML mappings," in *Proc. LDOW@ WWW*, 2016, pp. 1–5.
- [29] B. De Meester, T. Seymoens, A. Dimou, and R. Verborgh, "Implementation-independent function reuse," *Future Gener. Comput. Syst.*, vol. 110, pp. 946–959, Sep. 2020.
- [30] V. Presutti and A. Gangemi, "Content ontology design patterns as practical building blocks for web ontologies," in *Conceptual Modeling-ER*. Barcelona, Spain: Springer, 2008, pp. 128–141.
- [31] M. G. Skjæveland, H. Forssell, J. W. Klüwer, D. P. Lupp, E. Thorstensen, and A. Waaler, "Pattern-based ontology design and instantiation with reasonable ontology templates," in *Proc. WOP@ISWC*, 2017, pp. 1–15.
- [32] J. W. Klüwer, M. G. Skjæveland, and M. Valen-Sendstad, "ISO 15926 templates and the semantic web," in *Proc. Position paper for W3C Workshop Semantic Web Energy Industries; I, Oil Gas*, 2008, pp. 1–6.
- [33] D. P. Lupp, M. Hodkiewicz, and M. G. Skjæveland, "Template libraries for industrial asset maintenance: A methodology for scalable and maintainable ontologies," in *Proc. CEUR Workshop*. Aachen, Germany: Technical University of Aachen, vol. 2757, 2020, pp. 49–64.
- [34] *NumPy*. Accessed: Jan. 26, 2023. [Online]. Available: <https://numpy.org/>
- [35] The Apache Software Foundation. (2022). *Apache Arrow*. Accessed: Feb. 6, 2023. [Online]. Available: <https://arrow.apache.org/>
- [36] W. McKinney. (2019). *Introducing Apache Arrow Flight: A Framework for Fast Data Transport*. Accessed: Jan. 25, 2023. [Online]. Available: <https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/>
- [37] The Apache Software Foundation. (2022). *Apache Parquet*. Accessed: Jan. 26, 2023. [Online]. Available: <https://parquet.apache.org/>
- [38] D. Johnston. (Nov. 2020). *Don't Put Data Science Notebooks Into Production*. Accessed: Jan. 4, 2023. [Online]. Available: <https://www.Martinowler.com/articles/productize-data-sci-notebooks.html>
- [39] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of jupyter notebooks," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, May 2019, pp. 507–517.
- [40] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "What's wrong with computational notebooks? Pain points, needs, and design opportunities," in *Proc. CHI Conf. Human Factors Comput. Syst.* New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–12.
- [41] J. Krogstie, "Capturing enterprise data integration challenges using a semiotic data quality framework," *Bus. Inf. Syst. Eng.*, vol. 57, no. 1, pp. 27–36, Feb. 2015.
- [42] Dremio. (2022). *Dremio | The Easy and Open Data Lakehouse Platform*. Accessed: Jan. 26, 2023. [Online]. Available: <https://www.dremio.com/>
- [43] The Apache Arrow PMC. *Introducing ADBC: Database Access for Apache Arrow*. Accessed: Jan. 26, 2023. [Online]. Available: <https://arrow.apache.org/blog/2023/01/05/introducing-arrow-adbc/>
- [44] InfluxData. (2022). *Welcome to InfluxDB IOx: InfluxData's New Storage Engine*. Accessed: Jan. 26, 2023. [Online]. Available: <https://www.influxdata.com/blog/influxdb-engine/>
- [45] DuckDB Foundation. *DuckDB—An in-Process SQL OLAP Database Management System*. Accessed: Jan. 26, 2023. [Online]. Available: <https://duckdb.org/>
- [46] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics," in *Proc. Conf. Innov. Data Syst. Res. (CIDR)*, 2021, p. 17.
- [47] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Sebastopol, CA, USA: O'Reilly Media, 2017.
- [48] F. Abel, "Enabling advanced and context-dependent access control in RDF stores," in *Proc. Semantic Web*. Berlin, Germany: Springer, 2007, pp. 1–14.
- [49] A. Padia, T. Finin, and A. Joshi, "Attribute-based fine grained access control for triple stores," in *Proc. 3rd Soc. Privacy Semantic Web-Policy Technol. Workshop Co-Organised 14th Int. Semantic Web Conf. (ISWC)*, 2015, pp. 1–15.
- [50] OPC Foundation. (Nov. 2022). *OPC 10000-1 UA Part 1: Overview and Concepts*. Accessed: Dec. 13, 2022. [Online]. Available: <https://opcfoundation.org/developer-tools/documents/view/158>
- [51] *Communication Networks and Systems for Power Utility Automation—Part 7–3: Basic Communication Structure—Common Data Classes*, International Electrotechnical Commission, Geneva, CH, Standard IEC 61850-7-3:2010+A1:2020, 2020.
- [52] *Communication Networks and Systems for Power Utility Automation—Part 7-410: Basic Communication Structure—Hydroelectric Power Plants—Communication for Monitoring and Control*, International Electrotechnical Commission, Geneva, CH, Standard IEC 61850-7-410:2013/A1:2016, 2016.

- [53] *Wind Turbines—Part 25–2: Communications for Monitoring and Control of Wind Power Plants—Information Models* International Electrotechnical Commission, Geneva, CH, Standard IEC 61400-25-2:2015, 2015.
- [54] The Apache Jena Project. *Arq—A SPARQL Processor for Jena*. Accessed: Jan. 26, 2023. [Online]. Available: <https://jena.apache.org/documentation/query/>
- [55] E. Daga, L. Asprino, and J. Dowdy. *SPARQL Anything is a System for Semantic Web Re-Engineering That Allows Users to... Query Anything With SPARQL*. Accessed: Jan. 25, 2023. [Online]. Available: <https://github.com/SPARQL-Anything/sparql.anything>
- [56] The Apache Jena Project. *Apache Jena Fuseki*. Accessed: Jan. 26, 2023. [Online]. Available: <https://jena.apache.org/documentation/fuseki2/>
- [57] World Wide Web Consortium. (2013). *SPARQL 1.1 Protocol*. [Online]. Available: <https://www.w3.org/TR/sparql11-protocol/>
- [58] D. Chaves-Fraga, E. Ruckhaus, F. Priyatna, M. Vidal, and O. Corcho, “Enhancing virtual ontology based access over tabular data with Morph-CSV,” *Semantic Web*, vol. 12, no. 6, pp. 869–902, 2021.
- [59] D. Chaves-Fraga, J. Toledo, and L. Pozo. *Enhancing Virtual Kg Access Over Tabular Data With RML and CSVW*. Accessed: Jan. 20, 2023. [Online]. Available: <https://github.com/oeg-upm/morph-csv>
- [60] A. Swartz. *RDFLib*. Accessed: Jan. 26, 2023. [Online]. Available: <https://github.com/RDFLib/rdfliib>
- [61] T. Tanon. *Oxigraph/Oxigraph: SPARQL Graph Database*. Accessed: Jan. 26, 2023. [Online]. Available: <https://github.com/oxigraph/oxigraph/tree/main/lib/oxrdf>
- [62] Python Software Foundation. *Subprocess—Subprocess Management—Python 3.11.1 Documentation*. Accessed: Jan. 26, 2023. [Online]. Available: <https://docs.python.org/3/library/subprocess.html>
- [63] K. Sengupta, P. Haase, M. Schmidt, and P. Hitzler, “Editing R2RML mappings made easy,” in *Proc. 12th Int. Semantic Web Conf. (Posters Demonstrations Track)*, vol. 1035, 2013, pp. 101–104.
- [64] P. Heyvaert, A. Dimou, B. De Meester, T. Seymoens, A.-L. Herregodts, R. Verborgh, D. Schuurman, and E. Mannens, “Specification and implementation of mapping rule visualization and editing: MapVOWL and the RMLEditor,” *J. Web Semantics*, vol. 49, pp. 31–50, Mar. 2018.
- [65] Ontopic. *Ontopic Studio, a Low-Code Solution for Building Knowledge Graphs*. Accessed: Jan. 20, 2023. [Online]. Available: <https://ontopic.ai/en/ontopic-studio/>
- [66] M. G. Skjæveland and L. H. Karlsen. (Nov. 2022). *Terse Syntax for Reasonable Ontology Templates (stOTTR)*. Accessed: Jan. 20, 2023. [Online]. Available: <https://dev.spec.ottr.xyz/stOTTR/>
- [67] The Modbus Organization. (Apr. 2012). *MODBUS Application Protocol Specification V1.1b3*. Accessed: Jan. 25, 2023. [Online]. Available: https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf
- [68] T. Tanon. *Spargebra*. Accessed: Feb. 6, 2023. [Online]. Available: <https://github.com/oxigraph/oxigraph/tree/main/lib/spargebra>
- [69] *OXRDF*. Accessed: Feb. 6, 2023. [Online]. Available: <https://github.com/oxigraph/oxigraph/tree/main/lib/oxrdf>
- [70] D. J. Abadi, “Query execution in column-oriented database systems,” Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, MA, USA, 2008.
- [71] H. Bast and B. Buchhold, “QLever: A query engine for efficient SPARQL+Text search,” in *Proc. ACM Conf. Inf. Knowl. Manage.* New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 647–656.
- [72] C. Harrelson. (Sep. 2006). *Happy Trails With Google Transit*. Accessed: Mar. 29, 2023. [Online]. Available: <https://googleblog.blogspot.com/2006/09/happy-trails-with-google-transit.html>
- [73] L. Asprino. *Experiments/GTFS at Main · SPARQL-Anything/Experiments*. Accessed: Jan. 25, 2023. [Online]. Available: <https://github.com/SPARQL-Anything/experiments/tree/main/gtfs>
- [74] L. Asprino, E. Daga, J. Dowdy, A. Gangemi, and P. Mulholland, “Materialisation approaches for Façade-based data access with SPARQL,” *Semantic Web J.*, vol. 12, 2022. [Online]. Available: <http://www.semantic-web-journal.net/content/materialisation-approaches-fa%C3%A7ade-based-data-access-sparql>
- [75] World Wide Web Consortium. (Jul. 2017). *Shapes Constraint Language (SHACL)*. Accessed: Jan. 18, 2023. [Online]. Available: <https://www.w3.org/TR/shacl/>



MAGNUS BAKKEN received the bachelor's and master's degrees in natural language processing from the University of Bergen, in 2011 and 2014, respectively. He is currently pursuing the Ph.D. degree in computer science with the Norwegian University of Science and Technology. From 2014 to 2020, he was a software developer. His research interests include creating knowledge graphs and using knowledge graphs to support industrial applications.

• • •