Towards a Knowledge Access & Representation Layer

Kevin Angele[1,2,*], Umutcan Şimşek[1] and Dieter Fensel[1]

[1]*Semantic Technology Institute Innsbruck, University of Innsbruck, Austria*
[2]*Onlim GmbH, Vienna, Austria*

Abstract

Knowledge graphs integrate data from heterogeneous sources resulting in a very large set of statements to be stored and managed. Handling large amounts of data and supporting multiple use cases with probably conflicting requirements in a single knowledge graph is infeasible. To this end, we present our ongoing work on a "Knowledge Access & Representation Layer" on top of a knowledge graph. With Knowledge Activators in its core, the layer reduces the size to operate on, supports conflicting requirements, and allows to integrate external data dynamically. We mainly present the specifications and tasks of a Knowledge Activator as the core of the layer.

Keywords

Knowledge Graph, Knowledge Access, Knowledge Access & Representation Layer, Knowledge Activators

1. Introduction

Knowledge graphs integrate data from heterogeneous sources for powering intelligent applications. At a specific size of knowledge graphs, operations (like error detection, duplicate detection, or query answering) are hard to scale. Additionally, representing various points of view having different (probably) conflicting requirements on the underlying data is infeasible within a single knowledge graph. Besides, specific use cases require data from external services for evaluating a single request which should be integrated on the fly. This results in three main challenges: Handling the vast amount of statements (size), supporting various (conflicting) points of view, and dynamically integrating external data. Those challenges significantly influence generic applications designed to support multiple use cases.

This paper presents our ongoing work on a layer called "Knowledge Access & Representation Layer" on top of knowledge graphs, operating on use-case-specific subgraphs (views). Those views support different points of view and reduce the amount of data the operations need to handle. Additionally, context-specific data is dynamically integrated without affecting the underlying knowledge graph[1]. The main contribution of this paper is the introduction of Knowledge Activators as the core of the layer and drawing feature directions for the implementation of this layer.

SEMANTICS 2022 EU: 18th International Conference on Semantic Systems, September 13-15, 2022, Vienna, Austria
*Corresponding author.
kevin.angele@sti2.at (K. Angele); umutcan.simsek@sti2.at (U. Şimşek); dieter.fensel@sti2.at (D. Fensel)

[1]This will be part of the future work and is not addressed in this paper.

2. Exemplary Use Case

An exemplary use case is the German Tourism Knowledge Graph (GTKG). The GTKG integrates and curates data from all regional tourism marketing organizations in Germany, resulting in the integration of 16 heterogeneous sources. Currently, the GTKG contains around *31K Events*, *32K POIs*, and *5K Tours* accumulating to more than 23M statements[2]. The number of statements proliferates with more regional marketing organizations integrating their data into the GTKG.

Operations[3] on the GTKG become slower the larger the knowledge graph gets (*size challenge*). This can result in severe issues, as the operations must not interfere with a query answering operation. Those operations further decrease the performance of the query answering tasks and might cause temporary inconsistencies. Equally important, different regions have varying constraints on the underlying data. Also, custom inference rules are used to infer region-specific knowledge (*various points of view challenge*). Those (conflicting) points of view are not representable within a single knowledge graph. Especially when contextual knowledge from an application is needed, not necessarily belonging to the data in the overall knowledge graph.

Let us consider two intelligent applications recommending vegan restaurants and restaurants for meat-eaters. For the recommendations, each application requires a rule inferring a ranking score used to show the top-ranked restaurants. The rule for the vegan application infers a ranking score based on the variety of vegan dishes. Analogous, a restaurant's variety of meat dishes is essential for meat-eaters. Using both rules within a single knowledge graph is impossible as they infer conflicting ranking scores. It might not be tragic for a meat-eater to get a vegetarian recommendation, but the reverse situation must be avoided. Furthermore, both intelligent applications serving information about restaurants only need a tiny amount of data from the overall knowledge graph. When operating only on the relevant data, the number of triples to be considered for reasoning can be reduced from 23M to a few hundred thousand.

3. Knowledge Access and Representation Layer

The Knowledge Access & Representation layer (see Figure 1) acts as a middle layer between the applications consuming the knowledge graph and the knowledge graph itself. In this setup, the knowledge graph is treated as a data lake allowing it to be erroneous and incomplete. For the layer on top use-case-related subgraphs so-called *views* are extracted. Those views reduce the size of data that needs to be considered for the operations and allow various points of view. *Knowledge Activators* are at the core of the introduced layer operating on and storing those views. A Knowledge Activator consists of a Micro TBox defining the terminology, constraints, and rules and the subgraph definition used to extract the relevant subgraph from the knowledge graph. Besides, Knowledge Activators allow integrating context-specific data from external sources with the data contained in the view by using an *External data integrator*[4]. The flow from the applications to the Knowledge Activators is defined with the help of a control flow engine and a data flow connector. Handling the communication between the Knowledge Activators

[2]The latest statistic can be found on: https://open-data-germany.org/datenbestand/ (last access: 13-05-2022)
[3]Currently focused on error detection and duplication detection.
[4]This will be part of the future work and is not addressed in this paper.

and the underlying knowledge graph is done by a graph database connector.

This paper will focus on Knowledge Activators since they are the core of this layer.

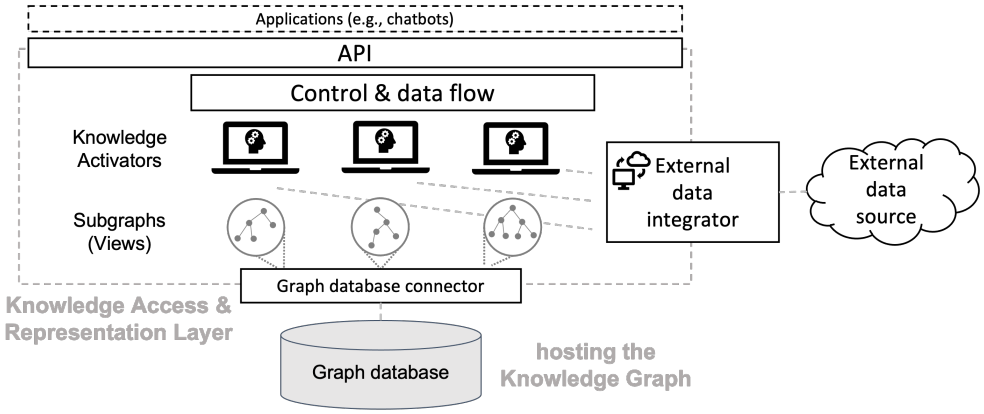

Figure 1: Overview of the Knowledge Access and Representation Layer

4. Knowledge Activators

Knowledge Activators operating on views are at the core of the Knowledge Access & Representation Layer. Extracting and hosting views, cleaning and enriching those, and integrating external data are the tasks a Knowledge Activator handles. Figure 2 gives an overview of the specifications and engines required to fulfill those tasks.

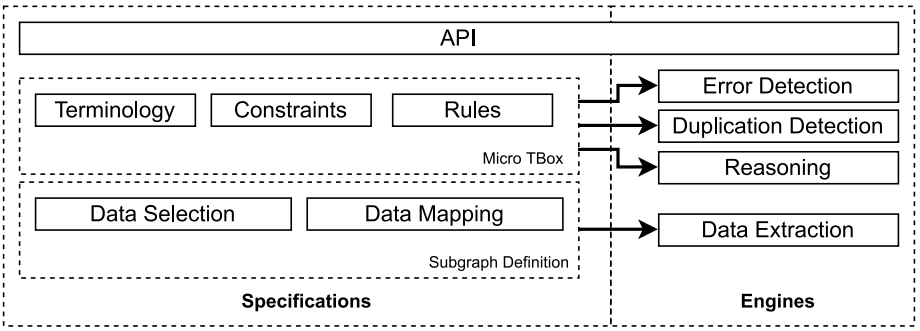

Figure 2: Specifications and Engines composing a Knowledge Activator

The specifications are grouped into *Micro TBox* and *Subgraph Definition* and form together with the extracted data a so-called *view*. A view is a use-case-specific subgraph with a context (Micro TBox) built on top of it. Use-case-specific implies that only data relevant for a given use-case is extracted from the underlying knowledge graph. A view reduces the size to operate on and supports various (conflicting) points of view by defining a specific view for each use case. After extracting the view from the knowledge graph, customizations can be applied to adapt the view according to the given requirements of the use case.

A *Micro TBox* contains the *Terminology, Constraints*, and *Rules*. The terminology defines types, properties, and the type hierarchy used within the view, not necessarily aligned with the underlying knowledge graph. It is possible to use completely different terminology, and even a different knowledge representation formalism is possible. Besides, constraints define specific requirements instances need to fulfill, and rules are used to infer new knowledge based on existing facts.

Subgraph Definition specifications are used for extracting the data for the view and are defined by Knowledge Engineers. Therefore, *Data Selection* specifies the relevant data from the underlying knowledge graph to be extracted (for example, by using GraphQL [1]). Mapping the terminology of the underlying knowledge graph to the terminology used within the view is done by a *Data Mapping* specification (e.g., using RML [2]). The subgraph definition specification is used by the *Data Extraction Engine* for extracting the data (for initializing a Knowledge Activator or on the fly).

Then, the data needs to be cleaned and enriched after extracting from the underlying knowledge graph because the knowledge graph can be erroneous and incomplete (we allow the underlying knowledge graph to be a data lake). Cleaning the data in the view (Knowledge Cleaning [3]) is about improving the correctness by identifying wrong assertions (called *error detection*) and correcting those (called *error correction).* We focus on error detection by applying integrity constraints to the data. Likewise, enriching the data in the view (Knowledge Enrichment [3]) targets the completeness of a view by integrating external sources and identifying duplicates the integration might cause. Furthermore, new knowledge can be inferred based on the existing facts by using rules.

An *Error Detection Engine* is used to identify wrong assertions using the terminology and constraints defined in the Micro TBox. Erroneous statements can be divided into *Syntactical Errors*, e.g., a URI contains whitespaces, and *Semantic Errors* where statements are not conform to the (Micro) TBox, e.g., the value of a property `age` is a `Text` instead of a `Number`. A validation report is produced by the Error Detection Engine containing all violations that need manual fixing.

The Duplicate Detection task aims to increase the completeness of a view by introducing lacking *sameAs* assertions between instances describing the same entity utilizing a *Duplicate Detection Engine*. Identifying and resolving duplicates is challenging, and many methods and techniques have been invented to tackle this issue [4]. In the end, the Duplication Detection engine provides a list of possible duplicates a Knowledge Engineer needs to check manually.

Inferring new knowledge using the rules is handled by a *Reasoning Engine*. When evaluating a request coming from an application, corresponding rules are evaluated, and inferred facts are included in the response. Besides, the reasoning engine integrates data from external services with the data from a view.

5. Conclusion and Future Work

This paper presented our ongoing work on the "Knowledge Access & Representation Layer", allowing the underlying knowledge graph to be a vast, erroneous, and incomplete data lake. For powering intelligent applications using the knowledge graph, Knowledge Activators extract

and host views, clean and enrich those, and cooperate with external data. This allows for use-case-specific constraints and rules. Additionally, the amount of data to operate on is much smaller, significant for the performance of the used engines.

So far, a first version of the graph database connector [1], the external data integrator [5], and the reasoning engine is implemented. For the error detection engine we further develop VeriGraph [6] and for deduplication as a service we further develop [4]. Furthermore, for defining the data flow we will use Apache NiFi[5] and an adoption of the Corinthian Abstract State Machine (CASM) [7] for the control flow engine.

Not addressed in this paper was the *dynamic data integration.* In the future, we will conceptualize and implement the cooperation of external data with data from the views on the fly using the *external data integrator* and the *reasoning engine*.

In the next steps, we first finalize the conceptualization to cooperate external data with data from the view. Then the existing implementations (*Database Extraction, Duplication Detection, Error Detection, External Data Integration*, and *Reasoning Engine*) are composed into the Knowledge Activators. Afterward, CASM will be adopted to fit our requirements for a control flow engine. After implementing the Knowledge Access & Representation Layer, an extensive evaluation will be conducted to showcase the performance improvements when operating on smaller subgraphs instead of the immense knowledge graph. In the end, the layer is used on top of the GTKG to support various applications.

References

[1] K. Angele, M. Meitinger, M. Bußjäger, S. Föhl, A. Fensel, Graphsparql: a graphql interface for linked data, in: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, 2022, pp. 778–785.

[2] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, R. Van de Walle, Rml: a generic language for integrated rdf mappings of heterogeneous data, in: Ldow, 2014.

[3] D. Fensel, U. Şimşek, K. Angele, E. Huaman, E. Kärle, O. Panasiuk, I. Toma, J. Umbrich, A. Wahler, Knowledge Graphs: Methodology, Tools and Selected Use Cases, Springer Nature, 2020.

[4] J. Opdenplatz, E. Huaman, E. Kärle, J. Umbrich, D. Fensel, Duplicate Detection as a Service (DDaaS), Technical Report D413y2, MindLab Project, 2019. URL: https://drive.google.com/file/d/1UfWwBLoxLmcdRYLudxJs90lq5E80bMsk/view?usp=sharing.

[5] E. Kärle, U. Şimşek, T. Gerrier, K. Angele, D. Fensel, KARL SWS Integrator, Technical Report D544y2, MindLab Project, 2019. URL: https://drive.google.com/file/d/1dxlVMvwiy9C8pn0IwJEQ6REltE-Qcy-M/view.

[6] K. Angele, O. Holzknecht, E. Huaman, O. Panasiuk, U. Şimşek, D. Fensel, VeriGraph: A verification framework for Knowledge Integrity, Technical Report D312y2, MindLab Project, 2019. URL: https://drive.google.com/file/d/1RudX-yt9JxomMb6OBCi4UD10vLtqWZBv/view.

[7] R. Lezuo, G. Barany, A. Krall, Casm: Implementing an abstract state machine based programming language, Software Engineering 2013-Workshopband (2013).

[5] https://nifi.apache.org/