2018 12th IEEE International Conference on Semantic Computing

Shipping Knowledge Graph Management Capabilities to Data Providers and Consumers

Omar Al-Safi[*], Christian Mader[†], Ioanna Lytra[†‡], Mikhail Galkin[†‡§], Kemele Endris[‡], Maria-Esther Vidal[†], Sören Auer[¶]
[*]RWTH Aachen University, Germany
omar.al-safi@rwth-aachen.de
[†]Fraunhofer IAIS, Germany
{Christian.Mader,Maria-Esther.Vidal}@iais.fraunhofer.de
[‡]Applied Computer Science, University of Bonn, Germany
{lytra,galkin,endris}@cs.uni-bonn.de
[§]ITMO University, Russia
[¶]German National Library of Science and Technology (TIB), Germany
soeren.auer@tib.eu

***Abstract*—The amount of Linked Data both open, made available on the Web, and private, exchanged across companies and organizations, have been increasing in recent years. This data can be distributed in form of Knowledge Graphs (KGs), but maintaining these KGs is mainly the responsibility of data owners or providers. Moreover, building applications on top of KGs in order to provide, for instance, analytics, data access control, and privacy is left to the end user or data consumers. However, many resources in terms of development costs and equipment are required by both data providers and consumers, thus impeding the development of real-world applications over KGs. We propose to encapsulate KGs as well as data processing functionalities in a client-side system called Knowledge Graph Container, intended to be used by data providers or data consumers. Knowledge Graph Containers can be tailored to the target environments. We empirically evaluate the performance and scalability of Knowledge Graph Containers with respect to state-of-the-art Linked Data management approaches. Observed results suggest that Knowledge Graph Containers increase the availability of Linked Data, as well as efficiency and scalability of various Knowledge Graph management tasks.**

I. Introduction

In our increasingly digitized world, data sharing and exchange between organizations in value chains, research collaborations, or other cooperation scenarios play a pivotal role. However, in many scenarios an effective and efficient data sharing is blocked by the fact, that

- either the data provider *loses control* over her data after shipping it to a cooperation partner – a partial mitigation can only be an expensive and cumbersome legal and contractual arrangement – or
- when keeping full control by only providing a remote access interface to the users – often associated with high latency –, query execution *costs* have to be fully covered by the data provider (or some remuneration negotiated) and it is hard to guarantee availability and performance to the data user.

As we learned in numerous workshops with more than 50 partner companies of the *Industrial Data Space Association*[1], keeping some level of control over the data – called *data sovereignty* – is a key requirement in industrial data sharing scenarios and currently the main obstacle for establishing data value chains in the industry. In many cases, cooperation partners should only gain access to a well defined fragment or usage access regime of the data. For example, a cooperation partner in a customer bonus program, should be enabled to access information about a specific customer (e.g., identified by name or member id), but should not be allowed to retrieve other sensitive customer data, like email and mailing addresses.

In physical value chains, containers play a key role in material, component, half product, and product exchange. Containers in most cases fulfill the function to secure, condition (e.g., cool/warm), observe, or provide access to their containment. In this work, we go the first step in the long term vision of realizing the concept of *Knowledge Graph Containers*, which is a key element in the Industrial Data Space reference architecture [1]. In order to realize the concept, we develop an approach which will allow for including data, security, access, and query processing functionality in a single artifact – the *Knowledge Graph Container*. Our approach is based on the recently emerging light-weight virtualization techniques and load balancing for scalable, high-performance query execution. As a result, we provide a novel data sharing and access paradigm, which balances costs and efforts differently between data provider and consumer than prior solutions (such as dumps, SPARQL endpoints, or Triple Pattern Fragments [2]). It potentially enables controlling data access even after data shipping, thus, it contributes to increased data sovereignty and, consequently, it better fulfills the requirements of industrial data value chains.

The main contributions of the current work include in particular:

[1]http://industrialdataspace.org

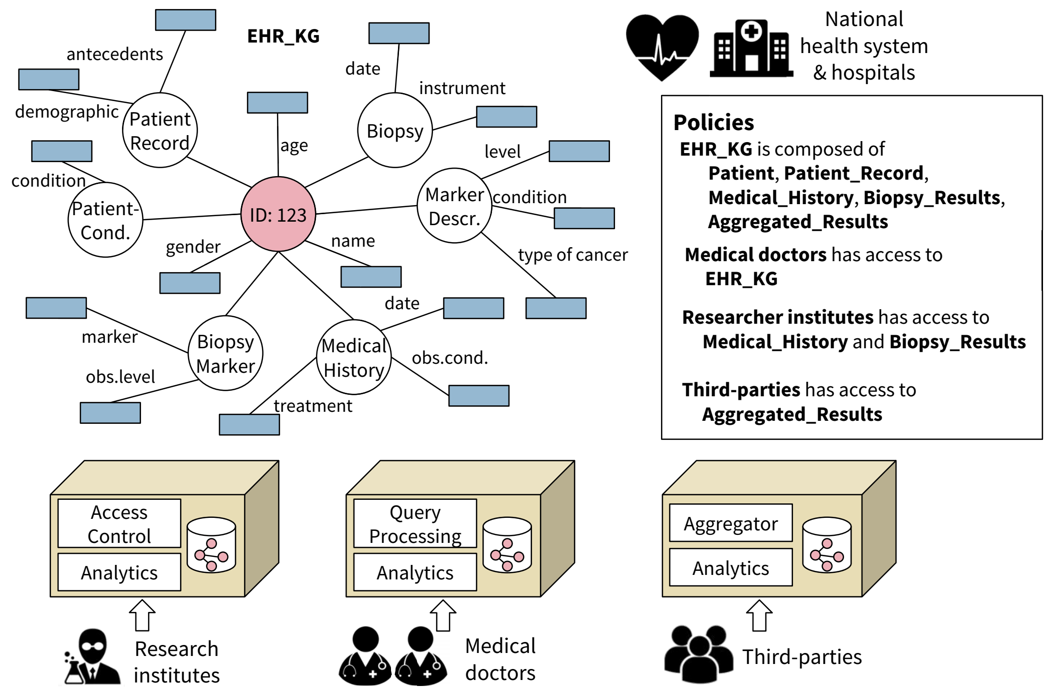


Fig. 1: **Motivating Example**. EHR_KG is a knowledge graph representing clinical data; it is composed of five knowledge graphs: Patient, Patient_Record, Medical_History, Biopsy_Results, and Aggregated_Results. EHR_KG is distributed to Research Institutes, Medical doctors, and Third-parties; access control and privacy policies state how portions of EHR_KG are accessed.

1) An approach for supplying data consumers with both data and processing capabilities, while enabling functionality for keeping control over the data and keeping efforts for data provider and consumer low.
2) An architecture to support distribution and usage of the approach supporting high-availability and scalability, while minimizing effort for both data provider and user.
3) An empirical evaluation where the Knowledge Graph Container approach is compared with state-of-the-art approaches with focus on efficiency. Results suggest that availability and scalability can be achieved without impacting resource consumption significantly.

This solution can be, of course, used by both data providers and consumers[2].

The remainder of the article is structured as follows. We motivate Knowledge Graph Containers using a real case scenario in the medical domain in Section II. In Section III, we introduce the problem tackled by the Knowledge Graph Container approach formally, and in Section IV, we introduce the Knowledge Graph Container Management System architecture. We perform an empirical evaluation of our approach and report on the evaluation results in Section V. Finally, we discuss the related work in Section VI and conclude with an outlook on future work in Section VII.

[2]In the remainder of the paper, data providers and consumers can be both users of the Knowledge Graph Container approach.

II. Motivation

Data in the form of Knowledge Graphs can be distributed to data consumers using different ways. For instance, data providers may choose to provide data dump files or Web interfaces with various data management capabilities to the end users. The first alternative is usually related to high installation and maintenance costs for the data consumers. In addition, there is no way for the data providers to introduce restrictions or any access control capabilities for the knowledge graphs they share. The second alternative provides, on the one hand, more flexibility to the data producer, on the other hand, the processing of the individual client requests can become very expensive in terms of CPU and memory and lead to scalability and availability issues.

In Figure 1, we illustrate a scenario of distributing clinical data to different end-users and organizations; various policies for accessing and processing this kind of sensitive data are considered. The national health system in collaboration with public hospitals produces big amounts of clinical data reporting personal information about patients, patient records, medical history, and biopsy results related to lung cancer; it aims at publishing these data as a knowledge graph, in order to support lung cancer research. However, sharing these data with various stakeholders poses several restrictions: medical doctors are allowed to access the whole datasets, while research institutes are entitled to use part of the knowledge graph, and other third-party organizations or users can use

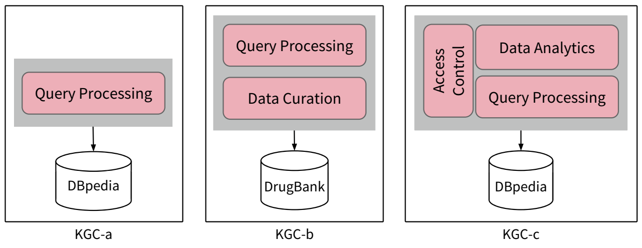


Fig. 2: **Examples of Knowledge Graph Containers (KGCs)**. Three KGCs with different data processing functionalities. KGC-a includes DBpedia and a query processing engine. KGC-b consists of DrugBank, and two data processing components: one for data curation and another for query processing. KGC-c encapsulates DBpedia, and query processing, analytics, and access control.

only aggregated functions on top of the data. These three solutions have to be distributed to the data consumers or other organizations that will play the role of data providers taking into consideration the following requirements: (1) the national health system and hospitals should keep control over the use of data they share and (2) the solutions should scale to the number of the end users. Until now, none of the approaches for publishing, sharing, and managing knowledge graphs can satisfy the aforementioned requirements.

III. Problem Statement

a) **Knowledge Graph Containers (KGCs):** a KGC encapsulates both (i) a knowledge graph and (ii) executable logic to perform knowledge management tasks over the knowledge graph. Thus, a KGC allows for shipping knowledge management from providers a means to support data consumers in making use of their data. For example, it is possible to embed data retrieval interfaces such as REST or SPARQL endpoints into a KGC. Furthermore, this method also gives data providers control over the usage of the data represented in a knowledge graph. Services embedded in KGCs can make use of, e.g., token-based authentication mechanisms so that access can be granted or withdrawn after a KGC has been transferred to the data consumer. Formally, a Knowledge Graph Container (KGC) corresponds to a pair KGC=$\langle D, En\rangle$, where D is a knowledge graph represented in the RDF data model, and En is a knowledge management engine able to execute knowledge management tasks over D. Figure 2 illustrates various knowledge management tasks that can be encapsulated in a KGC, and presents three different KGCs: i) KGC-a is composed of DBpedia and a query processing engine; ii) KGC-b provides not only query processing over DrugBank but also data curation tasks; and iii) KGC-c also encapsulates DBpedia, a query engine, and components for access control and data analytics.

b) **The Knowledge Graph Container (KGC) Approach:** The KGC approach provides data consumers not only with knowledge graphs, but also with knowledge management engines able to execute different knowledge management tasks

TABLE I: **Notation**. Symbols used in the Knowledge Graph Container Approach

Symbols	Description
KGC=$\langle D, En\rangle$	KGC encapsulating knowledge graph D and a knowledge management engine En
SKGC	Set of KGCs
KMT	$t_1,\ldots,t_m$ sets of knowledge management tasks
$F(.)$	Cost of evaluating a knowledge management task
LA	A load assignment of knowledge management tasks at KGCs
(KGC$_j$,KMT$_j$)	Load assignment of knowledge management tasks in KMT$_j$ to a KGC KGC$_j$
$\mathcal{F}(.)$	Cost of the load of a KGC
$\mathbb{F}(.)$	Maximum load cost among KGCs in a load assignment

over these knowledge graphs, e.g, access control, curation, or query processing. Moreover, the Knowledge Graph Container approach comprises load-balancing techniques able to adjust knowledge management to the computational resources available at a data consumer site. The following definitions state the core concepts required to formulate the problem tackled by the KGC approach; Table I summarizes the notation used to formalize the KGC approach.

c) **Load Assignment of Knowledge Management to KGCs:** Given SKGC={KGC$_1$,...,KGC$_n$} and KMT={t$_1$,...,t$_m$} sets of KGCs and knowledge management tasks, respectively. A function F: KMT $\rightarrow \mathbb{R}$, such that, F(t$_j$) represents the cost of executing task t$_j$. For instance, in KGC-c of Figure 2, SKGC corresponds to a set of replicas of KGC-c, i.e., KGCs that all have both the same DBpedia knowledge graph and knowledge management tasks KMT; KMT is composed of an engine able to perform query processing, data analytics, and access control over DBpedia. Moreover, values of $F(.)$ correspond to estimates of the cost of executing any of these tasks by the knowledge management engine of KGC-c, e.g., execution time, CPU usage, or I/O operations.

Knowledge management tasks in KMT are assigned to KGCs in SKGC; we name this assignment a *load assignment*. Formally, a load assignment LA is defined as follows:

- LA is a set of pairs representing knowledge management tasks in KMT assigned to KGCs, i.e.,

$$\text{LA} = \{(\text{KGC}_j, \text{KMT}_j) \mid \text{KGC}_j \in \text{SKGC and} \\ \text{KMT}_j \subseteq \text{KMT}\} \quad (1)$$

  where each (KGC$_j$,KMT$_j$) represents that knowledge management tasks in KMT$_j$ are executed at a KGC KGC$_j$ in a sequential order and only one at a time. We call (KGC$_j$,KMT$_j$) a *KGC load assignment* in the load assignment LA.
- $\mathcal{F}(.)$ is the *cost of a KGC load assignment* in LA, i.e., the cost of evaluating all the knowledge management tasks assigned to a KGC in LA. $\mathcal{F}$((KGC$_j$,KMT$_j$)) corresponds to the sum of the costs of evaluating all knowledge management tasks in KMT$_j$, i.e.,

$$\mathcal{F}((\text{KGC}_j, \text{KMT}_j)) = \sum_{\text{t}\in\text{KMT}_j} F(\text{t}) \quad (2)$$

- $\mathbb{F}(.)$ represents *the cost of evaluating the load assignment* LA, and is defined as the maximum *cost of the KGC load assignments* in LA, i.e.,

$$\mathbb{F}(\mathrm{LA}) = \mathrm{MAX}_{(\mathrm{KGC}_j,\mathrm{KMT}_j)\in\mathrm{LA}}\mathcal{F}((\mathrm{KGC}_j, \mathrm{KMT}_j)) \quad (3)$$

Consider a set SKGC=$\{\mathrm{KGC}_1,\mathrm{KGC}_2,\mathrm{KGC}_3\}$ with three replicas of KGC-c in Figure 2, and set KMT=$\{t_1,t_2,t_3,t_4\}$ of four knowledge management tasks. A load assignment LA of knowledge management tasks in KMT to KGCs in SKGC is the following:

$$\mathrm{LA} = \{(\mathrm{KGC}_1, \{t_1, t_2\}), (\mathrm{KGC}_2, \{t_3\}), (\mathrm{KGC}_3, \{t_4\})\} \quad (4)$$

Because knowledge management tasks assigned to a KGC, e.g., KGC_1, are executed sequentially and only one at a time, $\mathcal{F}(\mathrm{KGC}_1,\{t_1,t_2\})$ corresponds to the sum of the cost $F(.)$ of t_1 and t_2. However, KGCs in SKGC are executed concurrently; thus, the cost of $\mathbb{F}(\mathrm{LA})$ is the maximum cost of $\mathcal{F}(.)$ for the KGC assignments in LA.

d) **Load-Balancing in the KGC Approach:** The KGC approach attempts to identify in a set SLA, a load assignment LA with minimal cost $\mathbb{F}(.)$. This load-balancing problem is defined as the following optimization problem:

$$\begin{aligned}\operatorname*{argmin}_{\mathrm{LA}} \mathbb{F}(\mathrm{LA}) = \{\mathrm{LA}\ |\mathrm{LA} \in \mathrm{SLA} \text{ and} \\ \forall \mathrm{LA'} \in \mathrm{SLA},\ \mathbb{F}(\mathrm{LA}) \leq \mathbb{F}(\mathrm{LA'})\}\end{aligned} \quad (5)$$

The problem of KGC load-balancing is NP-complete even for two KGCs [3]. We devise an implementation of the KGC approach able not only to ship knowledge graphs from data providers to data consumers, but also knowledge management functionality. The KGC approach provides a KGC management platform (KGC Manager) to be installed by data consumers. The KGC management platform is able to manage different replicas of a KGC. Moreover, the KGC management platform balances the load of knowledge management tasks across replicas of a KGC, thus providing a practical solution to the problem of KGC load-balancing defined in Equation 5.

IV. The Knowledge Graph Container Management System Architecture

A. Managing Knowledge Graph Containers

A Knowledge Graph Container (KGC) Manager is a system able to realize the control and execution of KGCs at a data producer or consumer environment. Various implementations of KGC Managers are possible depending on the available resources of the data consumer, and the purpose for which the knowledge graph will be used. For instance, if a data consumer has limited hardware resources, she may configure a light-weight KGC Manager with a single instance of a KGC. If scalability is needed, she may utilize a KGC Manager capable of creating multiple replicas of a single KGC on multiple nodes so that load, e.g., CPU, memory, or network, can be distributed across these nodes. Likewise, implementations of KGC Managers may provide additional security and access control features.

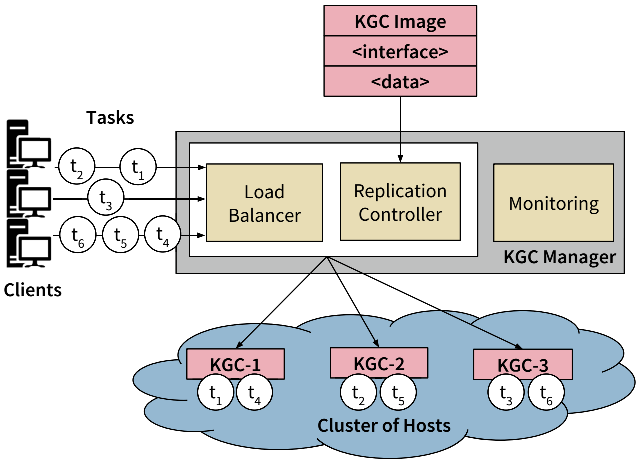


Fig. 3: **Knowledge Graph Container Manager**. It consists of a Replication Controller for creating multiple replicas (KGC 1-3) based on a KGC Image, a Load Balancer for distributing the incoming knowledge management tasks to a cluster of hosts, and a Monitoring component for logging resource usage and issuing alerts.

We focus on a KGC Manager tailored for scalability and availability; Figure 3 depicts the components of this KGC Manager. Given a KGC or KGC image, and a set of client tasks, the KGC Manager creates several replicas of the KGC image to distribute and balance the load of the tasks. Internally, the KGC Manager consists of three components, a *Load Balancer*, a *Replication Controller*, and a *Monitoring* component.

a) **Load Balancer:** Provides a solution to the problem of load-balancing defined in Equation 5. The Load Balancer receives a set of tasks $\{t_1,\ldots,t_n\}$ and distributes these tasks across a set of replicas of KGCs of a KGC image, in a way that the cost $\mathbb{F}(.)$ of answering all tasks is minimized. Cost can correspond, for instance, to total query execution time, total CPU usage, or I/O operations. The Load Balancer implements a Greedy algorithm that follows a round robin scheduling and assigns a task to the KGC with *least current load*.

b) **Replication Controller:** Creates a desired number K of KGC replicas from a KGC Image and deploys them to the data producer or consumer environment. A replication controller ensures that up to K replicas are up and available always. Thus, a task load can be distributed and balanced across the replicas. An estimate of available resources is utilized to determine the number of replicas that can be maintained by the data producer or consumer environment. If a replica fails, a new replica is created according to the estimate of available resources. Once the KGC Manager is shut down, all managed replicas are also shut down.

c) **Monitoring:** Logs resource usage and alerts data consumers in case of unexpected situations, e.g., whenever a KGC replica crashes. As such, these logs have a separate storage and lifecycle independent of a replica. Thus, accessibility of the application logs can be ensured for a period of time in order

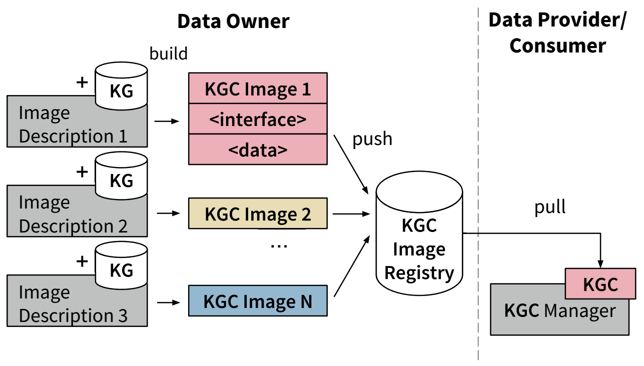


Fig. 4: **Creation and Distribution of Knowledge Graph Containers**. Data owners (left side) create and make available KGC Images, as well as push them to the KGC Image Registry. On the data provider or consumer side (right side), a KGC Manager pulls a KGC image from the Registry and creates a KGC replica.

to query or analyze the logs, regardless of the lifecycle of the replicas of a KGC image.

B. Creation and Distribution of Knowledge Graph Containers

As shown in Figure 4, data owners create KGC Images and push them to a KGC Image Registry where they are afterwards available for retrieval. Each KGC Image is built from an Image Description, KGC=$\langle D, En \rangle$, where the owner states: i) a knowledge graph D that the image will contain, and ii) a knowledge management engine En. In order to make use of a KGC Image, the data consumer needs to retrieve and install a KGC Manager (see Figure 3), which will pull the KGC Image from the KGC Image Repository and create a replica of the KGC image. When a KGC is started, the data consumer can start invoking the KGC knowledge management engine En against the knowledge graph D, e.g., to execute queries, enforce access control policies, or perform curation over the knowledge graph.

V. Empirical Evaluation

In our empirical evaluation, we focus on a specific type of KGCs: *KGC that contains a knowledge graph together with query processing capabilities*. In particular, we empirically study the efficiency of the KGC approach in processing SPARQL queries against DBpedia dataset. We compare its performance to the DBpedia public endpoint[3] and a Triple Pattern Fragment (TPF) [2] client. For this, we use a testbed of 24 queries[4] with different levels of complexity and selectivity against DBpedia version 2016-04. In addition, we study the benefits of the KGC Manager, i.e., the Load Balancer and the Replication Controller in the distribution of the workload to multiple KGCs, when an increasing number of queries from different users arrive to the system concurrently. The experimental configuration is as follows:

[3]http://dbpedia.org/sparql

[4]https://github.com/omarsmak/Linked-Data-Containers

A. Implementation

We implemented the KGC architecture on top of the Docker[5] ecosystem. The *KGC Image generation and provisioning* process on the data provider side involves three stages:

1) *Build*: The first stage is to build a KGC (Docker) base Image that is tailored to hold data of a specific type and includes an appropriate data processing engine for this data type. In this work, we use DBpedia version 2016-04 as the knowledge graph, and a SPARQL endpoint (Virtuoso) as knowledge management engine.
2) *Ingestion*: In the second stage, the data provider extends the KGC base Image from the first step by adding the actual dataset(s) that should be published, resulting in the final publishable KGC Image. This stage is performed by using the Docker build engine.
3) *Registration*: In the third stage, the newly created KGC Image is pushed to the data provider's KGC Image Registry, in our implementation to the Amazon EC2 Container Registry[6].

We developed a RESTful service in Python 3.7 that performs the last two steps by only requiring data owners to select the knowledge graph to be published. Selection of the appropriate KGC base image, creation of the publishable KGC Image, and uploading of the KGC Image Registry are performed by the service[7].

For implementing the *KGC Manager*, we used *Rancher Cattle*[8] along with *Rancher UI*[9], which is a Docker container orchestration engine that provides a large set of features such as container management, host management, auto scaling, container monitoring, and operations across containers. Rancher provides a docker-compose like configuration tool called Rancher-compose, which helps KGC to deploy containers on the hosts according to the appropriate scheduling polices, such as port conflicts and host tagging.

For *Load Balancing*, we stick to the default Rancher configuration which is using *HAProxy*[10] and the `roundrobin`[11] balancing strategy.

B. Infrastructure

Evaluation experiments are executed on *Amazon Elastic Compute Cloud* (EC2). We installed KGC Manager on a dedicated machine of type `m4.large` which comes with 2 CPU cores of 2.3 GHz and 8GB of RAM. For the cluster of hosts, we used four machines of type `m4.xlarge` which comes with 4 CPU cores of 2.3 GHz and 16GB of RAM; we made sure that multiple containers can run on this type of machine. To download and package our test dataset into a KGC image, we

[5]https://www.docker.com/

[6]https://aws.amazon.com/ecr/

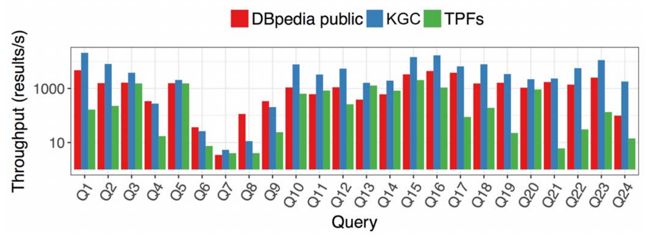
[7]Code is available at https://bitbucket.org/omarsmak/ids_container

[8]https://github.com/rancher/cattle

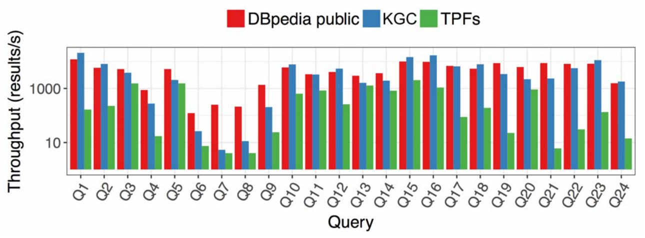
[9]http://rancher.com/

[10]http://www.haproxy.org/

[11]http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4.2-balance



(a) Throughput (results/sec) on Cold Cache



(b) Throughput (results/sec) on Warm Cache

Fig. 5: **Throughput (results/sec) on Cold and Warm Cache**. DBpedia public SPARQL endpoint, a single KGC, and a TPF client are compared for a Benchmark of 24 queries over DBpedia version 2016-04. For queries in KGC run in cold cache, the results for the public SPARQL endpoint correspond to the minimum throughput values measured whenever queries are run the first time. For queries in KGC run in warm cache, the results for the public SPARQL endpoint correspond to the maximum throughput values measured after running each query 20 times.

used a machine of type `m4.2xlarge` that comes with 8 CPU cores of 2.3 GHz and 32GB of RAM.

C. Metrics

a) **Execution Time:** Elapsed time between the submission of the query to KGC and the arrival of all answers. Time corresponds to absolute wall-clock system time as reported by the Python `time.time()` function.

b) **Throughput (results/sec):** Number of answers per second returned by successfully processed queries.

c) **Throughput (queries/sec):** Number of requests per second arrived at the system and processed successfully.

d) **Inverse Error Percentage (Inv.Error):** Inverse of the percentage of requests that fails with an error, i.e., Inv.Error corresponds to 100-ErrorPercentage.

e) **Received KB/sec:** Number of Kilobytes per second returned by successfully processed queries.

f) **Sent KB/sec:** Number of Kilobytes per second arrived at the system. We report on the inverse value, i.e., Inv.Sent KB/sec=$\frac{1}{SentKB/sec}$.

g) **Avg.Bytes:** Average amount of data measured in Bytes that is received during the execution of all the requests.

h) **CPU Usage:** Percentage of time CPU spent running the query processing processes of KGC; this does not include any process related to Linux kernel. The percentage corresponds to the percentage reported by the native Linux *top*[12] command.

i) **Block I/O Read:** Number of bytes read operations per second from disk by the processes of KGC. This corresponds to *Block I/O* metric that is reported by *docker stats*[13] command.

D. Results and Discussion

Figure 5a and Figure 5b report on the throughput (results per sec in log scale) of the DBpedia public endpoint, a single KGC, and a TPF client for our testbed of 24 queries (Q1–Q24), run on both cold and warm caches. In particular, we execute the SPARQL queries 20 times and compare the maximum and minimum throughput values measured. In general, both DBpedia public endpoint and KGCs perform better than TPFs by one to two orders of magnitude. We observe that in the case of cold cache, KGC outperforms DBpedia public and TPFs for 20 of the 24 queries, while the throughput of KGC in warm cache is better or comparable to the DBpedia public endpoint in the majority of the queries. Only for selective queries like Q6 (9), Q7 (19), and Q8 (16) DBpedia endpoint is significantly better while for some of the less selective queries such as Q1 (222.820), Q12 (335.454), and Q15 (117.571) KGC is the winner. However, the public endpoint of DBpedia is not able to return all results of the non-selective queries since the number of returned answers is restricted to 10.000; therefore, KGC is producing more complete results.

Figure 6 reports on the performance of the KGC Manager for one, two, three, and four KGCs. In particular, we compare these four configurations, for different number of user requests (i.e., 500 and 1000, respectively) with respect to Inv.Error, Throughput (results/sec), Avg.Bytes, Received KB/sec, and Inv.Sent KB/sec. Results suggest that considering more than one KGC replica enhances the performance of the KGC Manager if a large number of requests are posed.

Further, we study the resource consumption per KGC in case of four KGC replicas. Figure 7 shows the block I/O operations per KGC for a period of time of 2,5 hours, during which random requests arrive at the system. If only one KGC is serving requests the block I/0 operations for this KGC are increasing while, as expected, the load per KGC reduces when more KGC replicas are allocated by the Load Balancer. Moreover, the same behaviour is observed for the CPU usage, the CPU usage is shared among the KGC replicas – as can be observed in Figure 8. The behavior exhibited by the KGC Manager allows us to conclude that encapsulating both knowledge graphs and knowledge management functionality, as well as effectively balancing the load, provides a solution for knowledge graph availability and scalability.

[12]https://linux.die.net/man/1/top

[13]https://docs.docker.com/engine/reference/commandline/stats/

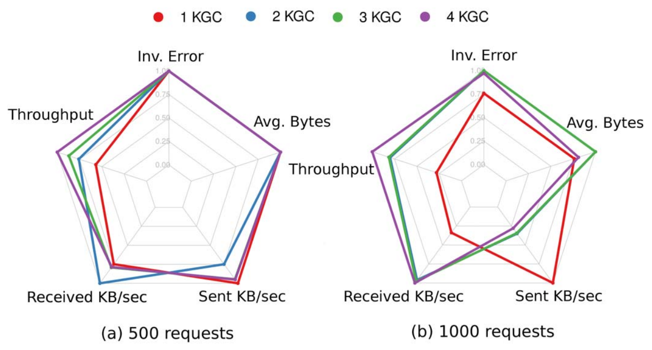


Fig. 6: **Performance of KGC Manager**. Performance is measured in terms: Inv.Error, Throughput (results/sec), Avg.Bytes, Received KB/sec, and Inv.Sent KB/sec; higher values are better. All the requests are submitted to the KGC Manager with one replica (1 KGC), two replicas (2 KGC), three replicas (3 KGC), and four replicas (4 KGC). (a) Results when 500 requests are posed to the KGC Manager. (b) Results when 1000 requests are posed to the KGC Manager. The configuration with four replicas exhibits better performance.

VI. Related Work

In the relational database world, the problem of efficient data replication has been in a focus for years. While almost every commercial RDBMS provides (at least limited) replication means, the academic community often aims at a particular replication characteristic overlooking the query completeness. Therefore, there exist numerous replication control techniques [4]. For instance, Cecchet et al. [5] propose *C-JDBC*, a middleware management system for database replication. C-JDBC orchestrates replicas in one virtual database and supports any RDBMS with JDBC access protocol. Lin et al. [6] present *SI-Rep*, a middleware replica control mechanism compatible with JDBC interface. Snapshot isolation preserves data consistency and ensures fault tolerance of the replicated collection. Similarly, we introduce efficient data replication and management for Knowledge Graph Containers.

Research on containerized shipment of datasets is still scarce. Arndt et al. suggest to containerize a knowledge base with a SPARQL endpoint and distribute it as a docker container to the data users [7]. Their methodology of Dockerizing Linked Data can be considered as an implementation of Knowledge Graph Containers. Still the aforementioned method is not designed to scale up for Big Linked Data applications with increased request load.

In addition to TPFs [2] and KBox [8], there exist approaches that address containerizing of data independent of its format. Such approaches often rely on physical data encryption inside a container. Lotspiech et al. [9] describe the *cryptolope* technology to manage access rights in a digital library. The technology employs encrypted data containers to which customers are able to purchase access permission and leverage its content by built-in query capabilities. Knowledge Graph Containers can be extended using similar techniques to ensure security and access control capabilities.

Sibert et al. [10] propose the *DigiBox* technology for e-commerce that tackles rights protection, data security, and interoperability in the scope of transaction data containers. However, the outlined approaches offer a full data encryption model that impedes the information exchange between data providers and consumers. Concentrating on the security aspects, those approaches miss a detailed analysis of possible impact on a company's infrastructure, containers interaction, and scalability that are covered by KGCs.

VII. Conclusions and Future Work

In this article, we presented the first step on our long term research agenda for fully realizing and evaluating the concept of Knowledge Graph Containers. In this work, we concentrated on the overall architecture employing light-weight virtualization for easing deployment and load-balancing for scalability. Efficiency of KGC approach was empirically compared with state-of-the-art approaches for a specific type of KGCs: KGCs containing a knowledge graph together with query processing capabilities. In addition, the scalability of our approach was evaluated. Empirical results suggest that the KGC approach increases availability of knowledge graphs and Linked Data, and provides a scalable and efficient platform to manage knowledge graphs at the data consumer side.

We experimentally showed the performance of the KGC approach for the query processing task over DBpedia; however, in the future, we plan to realize the KGC concept for knowledge graphs represented in different formats (e.g., relational, graph data), but equipped with knowledge capturing techniques to create their corresponding RDF knowledge graphs on demand. Another focus of our future work – which is not covered in the current article – will be experimenting with different techniques for realizing data security and access control, in order to ensure reliable usage guarantees and sovereignty for data owners. Security and Access Control can be seen as separate components, deployable in the KGCs.

We hope that ultimately the work towards KGC will contribute to establishing significantly more data value chains in the industry. As a result, shifting from the currently prevailing centralized data silo business models to more federated knowledge sharing business models will allow for leveraging the benefits of digitization for a larger number of businesses and in various business scenarios.

Acknowledgment

This project has received funding from the German BmBF for the project *InDaSpace*, the EU Horizon 2020 R&I programme for the Marie Skłodowska-Curie action *WDAqua* (GA No 642795), and CSA *BigDataEurope* (GA No 644564).

References

[1] B. Otto, S. Lohmann, S. Auer, J. Cirullies, and S. Wenzel, "Reference Architecture Model for the Industrial Data Space," Fraunhofer Society, Tech. Rep., 2017. [Online]. Available: http://dx.doi.org/10.13140/RG.2.2.17352.11529

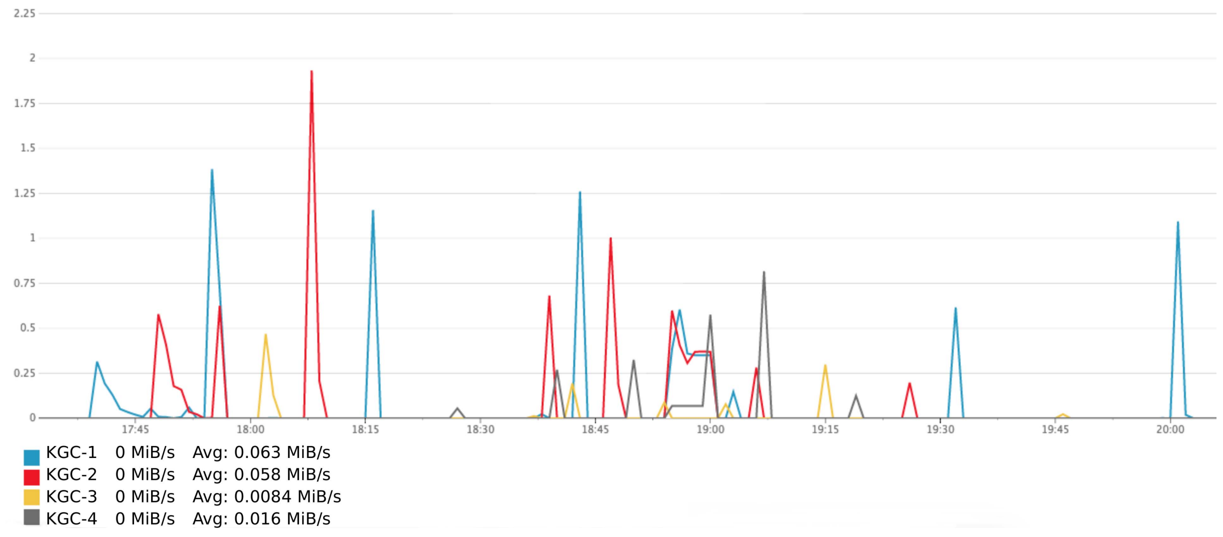


Fig. 7: **KGC Block I/O Read**. High KGC's block I/O read operations when there is only one KGC executing queries. Meanwhile the load reduces steadily when there are more KGC image replicas executing the same queries.

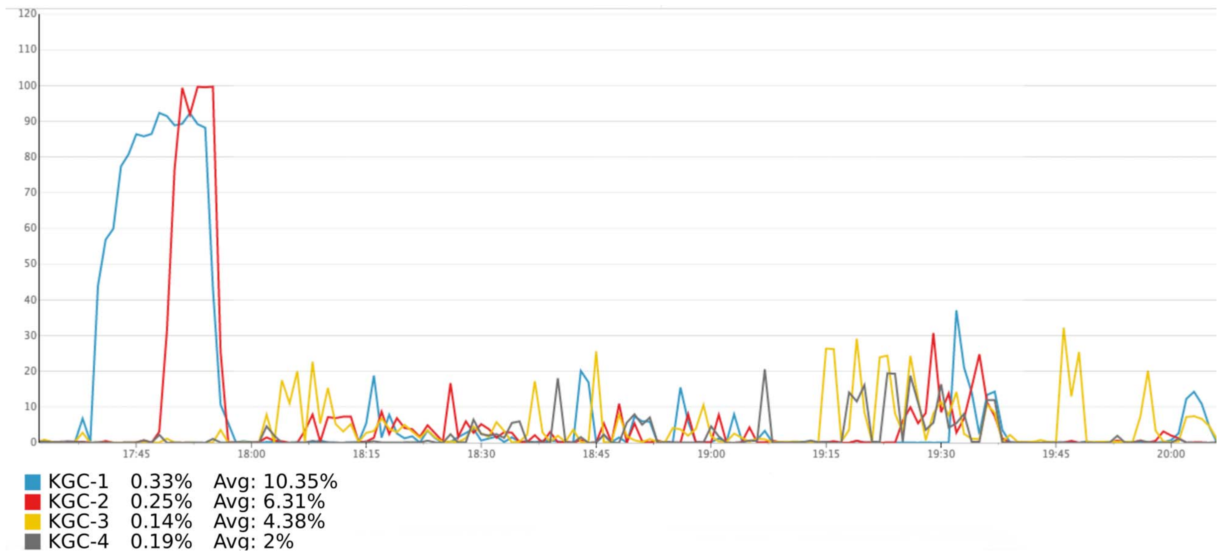


Fig. 8: **KGC CPU Usage**. High KGC's CPU load when there is only one KGC executing queries. Meanwhile the load reduces steadily when there are more KGC image replicas executing the same queries.

[2] R. Verborgh, M. V. Sande, O. Hartig, J. V. Herwegen, L. D. Vocht, B. D. Meester, G. Haesendonck, and P. Colpaert, "Triple pattern fragments: A low-cost knowledge graph interface for the web," *J. Web Sem.*, vol. 37-38, pp. 184–206, 2016.

[3] J. M. Kleinberg and É. Tardos, *Algorithm design*. Addison-Wesley, 2006.

[4] E. Cecchet, G. Candea, and A. Ailamaki, "Middleware-based database replication: the gaps between theory and practice," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, June 10-12, 2008*, 2008, pp. 739–752.

[5] E. Cecchet, M. Julie, and W. Zwaenepoel, "C-JDBC: Flexible database clustering middleware," in *USENIX Annual Technical Conference*, no. LABOS-CONF-2005-001, 2004.

[6] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris, "Middleware based Data Replication providing Snapshot Isolation," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, 2005, pp. 419–430.

[7] N. Arndt, M. Ackermann, M. Brümmer, and T. Riechert, "Knowledge base shipping to the linked open data cloud," in *Proceedings of the 11th International Conference on Semantic Systems, SEMANTICS 2015, Vienna, Austria, September 15-17, 2015*, 2015, pp. 73–80.

[8] E. Marx, C. Baron, T. Soru, and S. Auer, "KBox - Transparently Shifting Query Execution on Knowledge Graphs to the Edge," in *11th IEEE International Conference on Semantic Computing, ICSC 2017, San Diego, CA, USA, January 30 - February 1, 2017*, 2017, pp. 125–132.

[9] J. Lotspiech, U. Kohl, and M. A. Kaplan, "Cryptographic containers and the digital library," in *Verläßliche IT-Systeme*. Springer, 1997, pp. 33–48.

[10] O. Sibert, D. Bernstein, and D. V. Wie, "The DigiBox: A Self-Protecting Container for Information Commerce," in *First USENIX Workshop on Electronic Commerce, New York, USA, July 11-12, 1995*, 1995.