UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group DICE Group

Bachelor's Thesis

Submitted to the DICE Group Research Group
in Partial Fullfilment of the Requirements for the Degree of

Bachelor of Science

Basilisk – Continuous Benchmarking for Triplestores

by
Fabian Rensing

Thesis Supervisor:
Prof. Dr. Axel-Cyrille Ngonga Ngomo

Paderborn, May 22, 2022

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Abstract. Abstract

Contents

1 Introduction

introduce ts acronym

In the field of Semantic Web, knowledge graphs are an important structure to represent data and its relationships. To easily store and query the data in these knowledge graphs, some data structure or database is needed. The special kind of database developed to store knowledge graphs are called Triplestores.

Since knowledge graphs can contain huge amounts of data which can also be subject to many changes, Triplestores need to be able to handle many different workloads. Some scenarios need to handle huge amount of data being added, while others need to handle a lot of changes on the current data. To better test and compare Triplestores in these diverse scenarios, benchmarks are performed to allow an appropriate comparison between different Triplestores[21].

In general, Benchmarks are used to measure and compare the performance of computer programs and systems with a defined set of operations. Often they are designed to mimic and reproduce a particular type of workload to the system. In the context of Triplestores, a benchmark usually consists of creating a given knowledge graph on which multiple queries and operations are performed.

Often Triplestores are developed in long iterations and are bench-marked only in a late stage of such an development iteration. Today benchmarks and the evaluation of their results are usually done manually and bind developers time. Thus, performance regressions are found very late or never.

Several benchmarks for Triplestores have been proposed [21]. IGUANA is a benchmark-independent execution framework [10] that can measure the performance of Triplestores under several parallel query request. Currently the benchmark execution framework needs to be installed and benchmarks need to be started manually. Basilisk is a continuous benchmarking service for Triplestores which internally uses IGUANA to perform the benchmarks. The idea is that the Basilisk service will check automatically (continuously) for new versions of Triplestores and start benchmarks with the IGUANA framework. Further it should be possible to start custom benchmarks on demand. If a new version is found in a provided GitHub- or Docker Hub-repository, Basilisk will automatically setup a benchmark environment and starts a benchmarking suite.

This means that developers do not have to worry about performing benchmarks at different stages of development.

In this thesis we continue the development of the Basilisk platform and deploy an instance

to a publicly available virtual machine.

The thesis is structured as follows. In Chapter 2 we take a look at the state of the art of Triplestore benchmarking. Chapter 3 introduces the fundamental concepts and topics to understand this thesis. The chapter 4 describes the architecture use in the Basilisk platform.

2 Related Work

This chapter reviews the state of the art of Triplestore benchmarking.

Several benchmarks have been proposed and developed to test Triplestores. Many of these existing benchmarks focus on different goals and scenarios. Section 2.1 and 2.2 explain the different benchmark types used to benchmark Triplestores. Section 2.3 gives a short introduction to benchmark execution frameworks. Benchmarking in general is explained in section 3.2.1.

2.1 Synthetic Benchmarks

Synthetic benchmarks are benchmarks where the data is artificially generated. Often the generation is influenced by real world scenarios to generate data comparable to real world datasets[15]. These synthetic benchmarks have the advantage, that they can be generated to arbitrary sizes. The main point of criticism for synthetic benchmarks is that the generated data can easily become too abstract. Often the generated scenarios are not representative of a real world scenario [20].

The LUBM Benchmark[15] is a synthetic benchmark which focuses on the reasoning and inferencing capabilities of the Triplestores under test. The test data is located in the university domain and can be generated to arbitrary size. The benchmark provides fourteen extensional queries that represent and test a variety of properties.

Another synthetic benchmark is SP^{2}Bench[22]. The data generated stems from the DBLP scenario. The benchmark generation tries to accomplish that the key characteristics and word distributions are close to the original DBLP dataset. The provided queries are mostly complex and the mean size of the result sets is above one million[20]. They also test for SPARQL features like union and optional graph patterns.

The WatDiv suite generates a synthetic benchmarks and consists of multiple tools[7]. First the data generator which generates scalable and customizable datasets based on the WatDiv data model schema. The query template generator generates diverse query templates which will then be used to generate actual queries. The queries get generate with the query generator which instantiates the templates with actual RDF terms from the generated dataset. For each template multiple queries can be generated. The benchmark only focuses on SELECT queries that does not make use of the union and optional pattern features of SPARQL.

2.2 Benchmarks Using Real Data

Benchmarks using real data are benchmarks for which copies of real datasets and queries are used to perform a benchmarks. The real queries are often taken from query logs of Triplestores and the datasets are based on real datasets[18, 20].

FEASIBLE is a benchmark generation framework which generates datasets and queries from provided query logs[20]. This has the advantage that the data used for the benchmark could stem from queries about a specialized real world topics rather than an abstract synthetic model. FEASIBLE can also generate queries for the other SPARQL query types beside SELECT.

2.3 Benchmark Execution Frameworks

Benchmark execution frameworks, as the name suggests, help in the execution of database benchmarks. Their tasks are to load the data, execute the test queries and measure the defined metrics to evaluate the system under test.

Many benchmarks provide their own execution environments, which makes the comparison between benchmarks difficult. Often those environments are specialized for the given benchmark and are not easily interchangeable[10].

The next sections focus on benchmark-independent execution frameworks.

2.3.1 IGUANA

Iguana is a SPARQL, benchmark-independent execution framework[10]. The framework gets a dataset and a set of queries and operations as input and then uses the SPARQL endpoint of the Triplestore to load and update the data, and to perform the benchmark queries. It allows the measurement of the performance during loading and updating of data as well as parallel requests to the Triplestore. Iguana is independent of any benchmarks which allows it to run in different configurations and with various existing benchmarks and datasets. This includes synthetic benchmarks (2.1) and benchmarks based on real data (2.2). The benchmark process is highly configurable by passing a configuration file to the Iguana framework.

2.3.2 HOBBIT Framework

The Hobbit framework is a distributed benchmarking platform designed to be able to scale up benchmarking for big linked-data applications[19]. It is a big framework which needs to be deployed on a local cluster or online computing services like Azure[1] or AWS[2]. The deployment of the platform and deploying new benchmarks to the platform can be challenging for new users of the system[19]. The data for benchmarks has to be stored in docker containers Or it needs to be generated or downloaded before an benchmark, which increases the complexity of the system. The data is then send over message queues to the benchmarked system.

With the Basilisk platform we try to develop a specialized solution for continuous benchmarking of Triplestores which does not need the technical complexity present in the Hobbit framework. Implementing this functionality into the Hobbit framework would introduce another level of complexity to the system. Basilisk focuses on a smaller use-case of benchmarking SPARQL endpoints continuously with as little overhead as possible.

[1] `https://azure.microsoft.com/`
[2] `https://aws.amazon.com/`

3 Background

This chapter explains the fundamental topics required to understand this thesis.

In section 3.1 basic elements of the research area of semantic web are explained. In section 3.2 required topics from the field of software development are explained, which are needed to understand the development process of this thesis.

3.1 Semantic Web Topics

The following topics come from the research area of Semantic Web. Since this thesis focuses mostly on the implementation and deployment of the Basilisk platform, these topics are mostly introduced to give a basic understanding to the context in which the Basilisk platform will be used.

3.1.1 Knowledge Graphs

Knowledge Graphs are graphs intended to represent knowledge of the real world or smaller scenarios. The knowledge stored in Knowledge Graphs is modeled in a graph-based structure. Nodes represent entities which are connected by various types of relations, represented by labeled edges in the graph. This has the benefit to represent complex relations between different nodes and edges[17].

The simplest knowledge graph consists of three elements. The subject entity, the object entity and the labeled edge between them describing their relation. This atomic data entity is called triple. In figure 3.1 a simple example of a knowledge graph is shown.

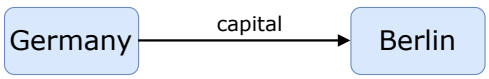

Figure 3.1: Simple Knowledge Graph

Since a graph structure is hard to store in a classic relational database a different type of storage is needed. The special kind of database developed to store knowledge graphs are called Triplestores.

3.1.2 RDF

The Resource Description Framework (RDF) is a framework for describing data and knowledge in a standardized way[5]. It is part of the W3C standard. The information is written down as subject-predicate-object triples, representing the basic structure that is also present in Knowledge Graphs (3.1.1). The elements of those triples can be IRIs (internationalized resource identifiers), blank nodes or datatyped literals.

RDF graphs can be encoded with different syntax styles. A popular syntax is TURTLE [6] which is a compact way of writing down a RDF graph structure. Using the example of section 3.1.1, the knowledge graph would be represented with the TURTLE syntax seen in figure 3.2. The first two lines of the TURTLE document define abbreviations for the used IRIs so that the triple in line three is more readable.

```
1 @prefix dbr: <http://dbpedia.org/resource/> .
2 @prefix dbo: <http://dbpedia.org/ontology/> .
3 dbr:Germany dbo:capital dbr:Berlin .
```

Figure 3.2: Example of an RDF graph in TURTLE syntax.

3.1.3 Triplestore

Triplestores are a special kind of database developed to easily store and access knowledge graphs through queries. Example of Triplestores are Tentris[9], GraphDB[1], Virtuoso[2], or Jena TDB[3].

This thesis focuses on Triplestores that have a SPAQRL endpoind and accept SPARQL queries, since the used benchmark framework Iguana is using the SPARQL endpoint to perform benchmarks (see section 2.3.1).

3.1.4 SPARQL

SPARQL (SPARQL Protocol and RDF Query Language)[16] is a query language for manipulating and retrieving RDF data stored in Triplestores. Just like RDF, SPARQL is part of the W3C recommendations for technologies in the semantic web.

The syntax for SPARQL queries looks similar to the SQL syntax, since its main parts are also a `SELECT` clause stating which variables to query for, following by an `WHERE` clause giving restrictions and conditions.

Queries can contain optional graph patterns, conjunctions, disjunctions, as well as aggregation functions. These extension can help formulate more complex queries.

Following the example from section 3.1.1 and 3.1.2 there are two example SPARQL queries in figure 3.3. Executed against the DBPedia SPARQL endpoint[4] the following results can be found:

The first example query requests the variable which matches the `WHERE` clause searching for the capital of Germany, which is `dbr:Berlin`. The second query requests all relationships that can be found between Germany and Berlin, which will return `dbo:capital`, which we expected, but also `dbo:wikiPageLink`, which means that there is a link from the Wikipage of Germany to the Wikipage of Berlin.

[1] `https://graphdb.ontotext.com/`
[2] `https://virtuoso.openlinksw.com/`
[3] `https://jena.apache.org/documentation/tdb/`
[4] `https://dbpedia.org/sparql`

```
1  PREFIX dbr: <http://dbpedia.org/resource/>
2  PREFIX dbo: <http://dbpedia.org/ontology/>
3
4  SELECT ?capital
5  WHERE {
6          dbr:Germany dbo:capital ?capital .
7  }
8
9  ---
10
11 SELECT ?relation
12 WHERE {
13         dbr:Germany ?relation dbr:Berlin .
14 }
```

Figure 3.3: SPARQL query examples

3.2 Software Development

The following topics can be grouped under the field of software development. For the topic of benchmarks (section 3.2.1) we focus on database benchmarks and especially Triplestore benchmarks, since this is the main task of the Basilisk platform. The sections Microservice and Microservice Architecture (3.2.2, 3.2.3) explain the basic idea and concept of the microservice architecture style. In the sections RabbitMQ and Spring (3.2.4, 3.2.5) we give a short introduction and description of the main technologies that are used for the development of the Basilisk platform.

3.2.1 Benchmark

Benchmarks for databases consist of a dataset and a set of operations or queries which will be performed on the dataset. These operations are designed to simulate a particular type of workload to the system. The goal of a benchmark is to measure different metrics for a better comparison between various systems. Metrics used for databases and Triplestores are e. g., number of executed queries and queries per second[2].

A distinction is made between micro and macro benchmarks. Micro benchmarks focus on testing the performance of single components of a system. Macro benchmarks test the performance of a system as a whole. The benchmarks performed by the Basilisk platform will only be macro benchmarks.

3.2.2 Microservice

A microservice is an independently deployable piece of software that only implements functionalities that are closely related to the main task of the service [11]. All Microservices can be individually deployed and managed. They interact via messages through a defined protocol with other services. The idea is that individual microservices can be combined like modules to create any desired complex software.

3.2.3 Microservice Architecture

A microservice architecture is a way of designing a software application as a set of microservices which interact with each other to provide the designed functionality [11, 3]. The functionality of the application gets split up into microservices which interact only through a defined message protocol. This allows for a distributed system in which the individual service could be implemented in different programming languages and also could be located on different servers.

3.2.4 RabbitMQ

RabbitMQ is an open-source message broker that supports different messaging protocols like MQTT, STOMP and AMQP. The system supports a variety of asynchronous messaging techniques e. g., delivery acknowledgment, flexible routing[4].

In the context of the Basilisk platform we only need the most basic functionalities of message queues with a single producer and a single consumer. Since RabbitMQ is a widely used message broker, the Spring framework (3.2.5) already comes with the needed libraries to work with the RabbitMQ system.

3.2.5 Spring and Spring Boot

Spring[5] is a widespread open-source Java framework which facilitates the development process for various kinds of java applications and systems.

Spring Boot[6] is an extension to the Spring framework that follow the convention-over-configuration design paradigm. This means that the implementation of applications has to follow common design conventions that replace a need for configuration files for many standard scenarios. Spring Boot also comes with preconfigured standard libraries for the Spring platform to ease the development for many standard applications like web-apps or microservices.

Spring and Spring Boot come with different annotations to decorate classes and methods that configure these automatically and tell the Spring framework how to handle and interact with their objects.

The Spring framework and Spring Boot use different software design conventions to structure the code and classes. The package structure found in the source code of the Basilisk microservices is influenced by Spring Boot and tries to represent those different design conventions.

3.2.6 Software Design Patterns

Software design patterns are used by developers to structure code. The idea is that code of different projects share reusable structures of best practices for commonly occurring implementation problems. Further more patterns are used to improve the understanding and readability of the code[12].

As stated in section 3.2.5 Spring Boot requires the usage of different software design paradigms and patterns which give the source code a common structure.

The following sections introduce design patterns which are used in the source code of the Basilisk platform.

Repository Pattern

The repository pattern is used to abstract the interaction to a data storage. This data storage can be an relational database or a data storage of any kind[14, 8].

The business logic of an application only interacts with a so called repository without actually knowing which kind of data storage is used. The repository then handles the storage and access of objects with the data storage.

This has the advantage that the actual storage system can easily be changed. When the storage system is changed, only the repository class needs to be adjusted. The business logic can still use the same interface of the repository class to access its data.

[5] `https://spring.io/`
[6] `https://spring.io/projects/spring-boot`

Domain Driven Design

Domain driven

fill

3.2.7 Stateful / Stateless Microservices

fill

Section about Docker deployment?

4 Approach

In this chapter we give an overview of the current software architecture on which the Basilisk platform is build.

The purpose of the Basilisk platform is to provide an easy way to continuously perform benchmarks on Triplestores. Triplestores are often developed in teams who collaborate in Git repositories. Releases of those Triplestores are then published on GitHub or as a Docker image on Docker Hub. The idea for the Basilisk platform is to offer the possibility to automatically perform benchmarks for a new Triplestore release.

Benchmarks are also relevant during the development process. An benchmark performed automatically for e. g. a new pull request is a good way to estimate if a newly developed feature will impact the performance of the Triplestore before the changes are merged.

On the Basilisk platform a user can register a Triplestore for a continuous benchmark by setting up a hook to the repository on GitHub or Docker Hub. The repository will then be observed by the Basilisk platform. If there is a new release of the Triplestore, Basilisk will generate a new benchmark job. This benchmark job will then be executed by fetching and building a new docker container containing the newest release of the Triplestore. On this container the benchmark will be performed. The measured results of the benchmark will be stored in a Triplestore and are then available through the web frontend for review.

The basic architecture pattern of the Basilisk platform is the microservice architecture (see chapter 3.2.3 for a short description). This means that the platform is divided into multiple services on which the workload and the different tasks are divided. The services could be run on different hardware systems and they interact with each other via a message queue system.

4.1 Programming Language and Frameworks

All services of the Basilisk platform are implemented with Java and are using the Spring Boot framework. The services use Java version 17 and Spring Boot version 2.6.5.

The package structure used for the implementation of the services is similar in all three services. It is strongly influenced by the structure recommended for the Spring Boot framework. The Spring and Spring Boot framework are shortly described in chapter 3.2.5.

4.2 Main Services

The next sections explain the three main services, namely Hooks Checking Service (section 4.2.1), Jobs Managing Service (section 4.2.2), and Triplestore Benchmarking Service (section 4.2.3).

This explanation follows the flow of actions that happen while configuring a continuous benchmark and the actions that happen when a benchmark is initiated.

The explanations are based on provided diagrams, code review and analysis, and information provided by former developers of the project.

Figure 4.1 gives an overview of the three microservices of the Basilisk platform. It shows the most important messages send between the services and the interactions with GitHub and Docker Hub.

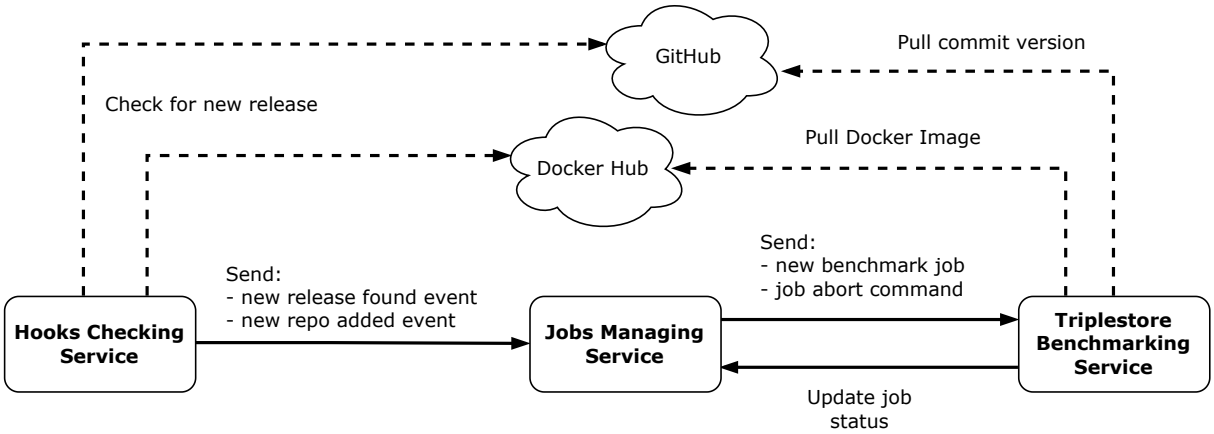

Figure 4.1: Overview of the three microservices

4.2.1 Hooks Checking Service

The main task of the Hooks Checking Service (HCS) is to observe GitHub and Docker Hub repositories of Triplestores for new releases or changes.

When a user wants to set up a new continuous benchmark, the HCS needs to be informed which repository (GitHub or Docker Hub) has to be observed for changes. This happens through REST API calls to the HCS providing the repository name and owner. The HCS will then create a hook for the repository to get notice about changes. A hook is in general a piece of code or software that attaches itself to a software component to intercept messages and react to those messages, e. g., with function calls. In the case of the HCS the hooks can be seen as bookmarks for the repositories. Each hook stores the latest known version of an repository. The service will query the saved repositories regularly and compare their current version to the version stored in the hook.

When the HCS notices a new release for a repository, it updates the corresponding hook to the newest version. Then it sends a message about the new version to the Job Request Queues from which the Jobs Managing Service retrieves the message.

API and Messaging

The HCS is controlled by the user over a REST API.

The continuous checking of the repositories can be started and stopped over a REST endpoint. The other most important endpoints are for adding and deleting GitHub and Docker Hub repositories. Figure 4.2 shows these endpoints.

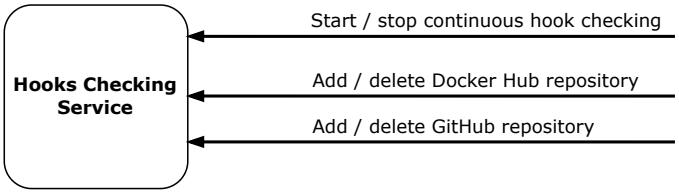

Figure 4.2: REST API of the Hooks Checking Service

The communication between the HCS and the Jobs Managing Service is done over RabbitMQ (3.2.4) messages, over the Job Request Queues. The messages contain different events that can occur in the HCS. For example an event is send when adding or deleting a repository, or a new release is detected.

4.2.2 Jobs Managing Service

The main task for the Jobs Managing Service (JMS) is to create benchmark jobs, when a new release was found by the HCS. Other important functionality of the JMS is the management of configurations needed for the benchmarks. Lastly the JMS manages the status for running and pending jobs send to the Triplestore Benchmarking Service.

There are three configuration types needed. First the platform needs the configuration for the Triplestore. This configuration include for example the SPARQL endpoint as well as the user and password for the connection to the endpoint. This is needed by the Iguana framework to properly connect to the Triplestore under test[1].

Secondly the platform needs configurations for datasets and query configurations. The dataset configuration simply consists of the dataset name and the URL for the location of the dataset. The query configuration consists similarly of a name for the queries and the URL for the location of the query file.

These configurations are added over the REST API of the JMS.

When the HCS sends an event regarding a new release of a repository, the JMS will create benchmark jobs for the new release. A benchmark job consists of the current version of the repository, a query configuration and a dataset. For each event multiple benchmark jobs can be created. For each query configuration and dataset one benchmark job will be created.

These benchmark jobs will then be send to the Triplestore Benchmarking Service over the Benchmark Job Queue.

The management of the running and pending benchmark jobs is done over the REST API of the JMS. When an endpoint is triggered, e. g., to abort a running job, the JMS sends an event to the Triplestore Benchmarking Service.

API and Messaging

The JMS communicates with the HCS and the Triplestore Benchmarking Service over RabbitMQ message queues. The service receives repository events from the HCS and sends benchmark job events to the Triplestore Benchmarking Service over the Benchmark Job Queue (BJQ).

Interaction of the user is handled over the REST API. The API offers endpoints for adding and deleting the different configurations of Triplestores, datasets and queries A second set of endpoints are for querying the job status of running and pending jobs, and for stopping individual jobs. Figure 4.3 shows these endpoints.

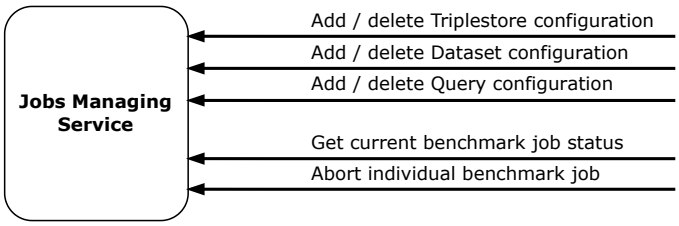

Figure 4.3: REST API of the Jobs Managing Service

4.2.3 Triplestore Benchmarking Service

The Triplestore Benchmarking Service (TBS) executes the benchmark jobs send by the JMS and saves the benchmark results to the Job Storage Triplestore.

To execute a benchmark the service needs a running instance of the Triplestore under test on which the benchmark will be executed. This instance is build from the information and configurations provided in the benchmark job. The TBS will query the provided repository (GitHub or Docker Hub) for the version specified in the job.

If the repository is from GitHub, the TBS downloads the source code for the provided commit and searches for a Dockerfile. It then builds and runs a Docker container from that Dockerfile.

If the repository is from Docker Hub, the TBS pulls the image with the provided tag. It then runs the image as a Docker Container.

After starting the Docker Container the TBS starts the Iguana framework. Iguana will perform the benchmark with the provided configuration from the benchmark job.

When the benchmark is finished the results are written to the Job Storage Triplestore

API and Messaging

The TBS has no REST API. The service is controlled through the JMS by events send over RabbitMQ.

The events received from the JMS are new benchmark jobs and pause or abort commands for running benchmark jobs. The TBS sends short events containing the status of benchmark jobs, e. g., a job has started, or it has finished and the results are uploaded to the Job Storage Triplestore.

4.3 Basilisk Frontend

The Basilisk platform can be extended with a web frontend. The frontend is implemented using JavaScript and the JavaScript Framework Vue.js

The idea is that the frontend functions as a graphical interface for the REST APIs of the three services explained in section 4.2. The user can setup new repositories, Triplestores and datasets. Further the user can request information about current benchmark jobs, abort jobs or remove pending jobs. Lastly the frontend can request and visualize the benchmark results stored in the Job Storage Triplestore.

4.4 Architecture Review

In this section we review the architecture of the three services of the Basilisk platform. We point out possible problems with current implementations and list missing implementations that need to be added.

4.4.1 Code Refactoring

During the code analysis some inconsistencies in the code style and duplicate code snippets have been found. In other parts the code structured differs to the design patterns recommended for the Spring and Spring Boot framework.

In general an in-depth code refactoring is recommended to increase readability and maintainability of the source code.

4.4.2 Management of Repositories and Configurations

Currently the observed repositories are managed and stored in the HCS while the configurations for the Triplestores are managed and stored in the JMS. This makes it difficult to internally link a repository to a Triplestore configuration, since they are stored in different services.

The current implementations tries to solve this problem, by sending events about repository creations from the HCS to the JMS. This results in the duplication of the repository storage in both services. This contradicts the idea of microservice, which should be separated as much as possible from each other.

Therefore we recommend restructuring the management of repositories.

4.4.3 Creation and Management of Benchmark Jobs

When a new release is found by the HCS, the Jobs Managing Service will create and manage benchmark jobs which will be executed by the TBS. Currently the JMS will create multiple jobs. For each query file a job will be created for each dataset. This means that each dataset is mixed with each query file, which could lead to queries executed on the wrong datasets.

A benchmark should only use a defined pair of a matching query file and dataset. Therefore the logic for creating the benchmark jobs needs to be changed, as well as the data model for storing the benchmark jobs.

4.4.4 Data Model Restructure

The JMS manages and stores the different configuration types needed for a benchmark job.

The configurations are stored in an internal database. Figure 4.4 shows the current database schema.

The schema has logical errors and is in parts incomplete. Following we list some inconsistencies and possible problems that could arise:

- The only way to identify an repository as GitHub or Docker Hub repo is to check in the Triplestore configuration. If a repository is wrongly assigned to the false type, the resulting benchmark job would not be executable, because GitHub and Docker Hub needs to be handled differently during a benchmark as explained in section 4.2.3.
- Each Triplestore configuration can have exactly one GitHub or Docker Hub repository. This means for every repository a new Triplestore configuration needs to be added. There would be less duplicate configurations if multiple repositories could point to the same configuration. For example, a hook could be set up to observe a GitHub repository for

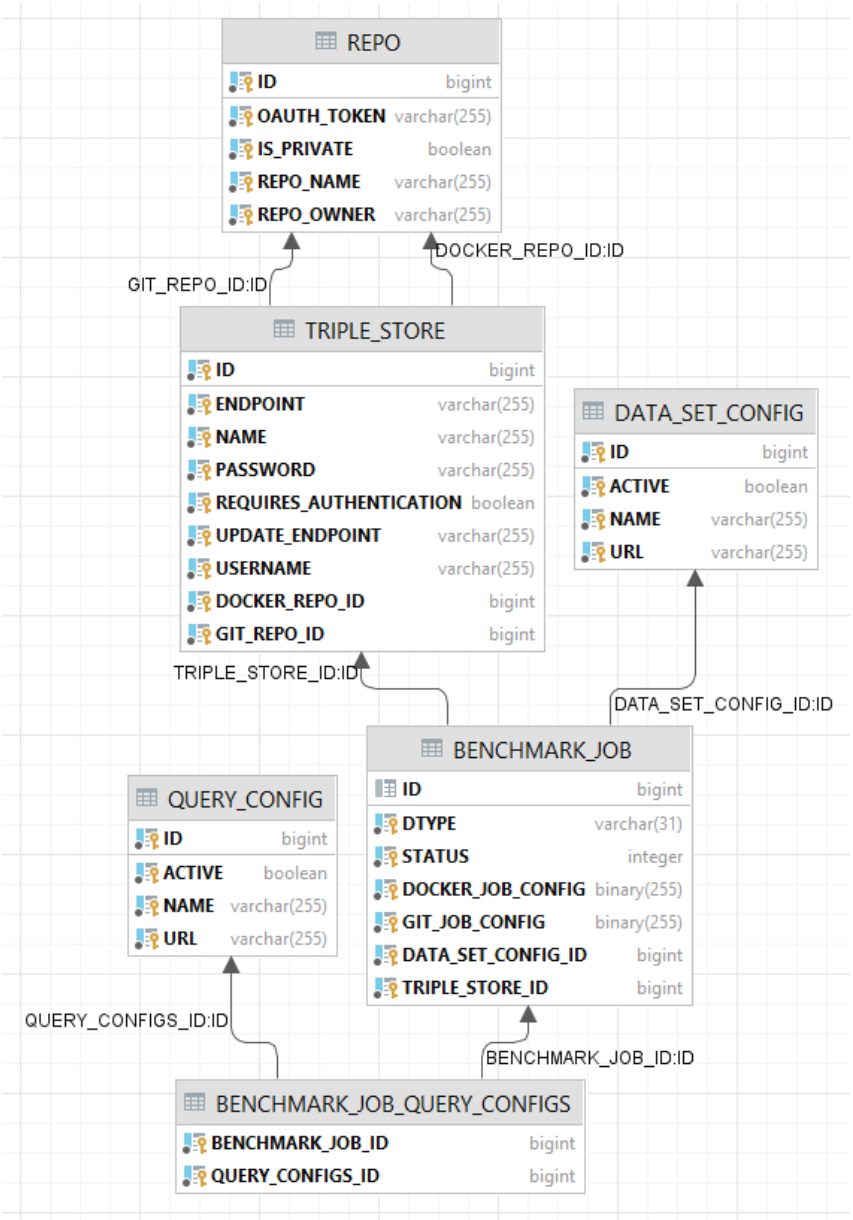

Figure 4.4: Diagram of the current database schema used in the JMS

new releases and another hook could be set up to observe the same GitHub repo for pull requests. In this case both hooks should use the same Triplestore configuration, since it is the same Triplestore which gets benchmarked.

- As explained in section 4.4.3, the creation and storage of benchmark jobs needs to be changed. Currently the data model structure for benchmark jobs, datasets and query files is too complicated and not easy to understand. Since the creation process of the jobs needs to be changed, the data model will also need changes and a cleanup of the model relationships.

Therefore the data model for the JMS needs a restructure to better cover real world requirements.

4.4.5 Missing Implementations

The Basilisk platform is not yet fully implemented. After reviewing the source code the following overview was created:

Hooks Checking Service

The implementation of the Hooks Checking Service is quite complete. Small additions have to be implemented.

- The REST endpoints for deleting GitHub and Docker Hub repositories needs to be added.
- Currently Pull Requests for GitHub repositories can not be observed.

Jobs Managing Service

The implementation of the Jobs Managing Service is mainly missing the REST API and some internal logic. The following REST endpoints have to be added:

- Adding / removing Triplestore configurations
- Adding / removing benchmark configurations
  - Adding / removing dataset configurations
  - Adding / removing query configurations

Since the JMS also manages the running and pending benchmark jobs, the REST API and internal logic for managing these jobs needs to be implemented too.

- List running / pending jobs and their status
- Aborting a benchmark job

Triplestore Benchmarking Service

The implementation of the Triplestore Benchmarking Service currently contains only a few classes for the data models, and simple structures of service classes. Big parts of the logic still needs to be implemented.

The existing classes are mainly for storing and manipulating data models, configurations and basic message queue interactions. These classes do not carry much functionality.

The main functionality needs to be implemented. This consists of setting up the Docker containers which contain the Triplestores for benchmarking:

- Pulling Code from GitHub
- Pulling images form Docker Hub
- building Docker containers from Dockerfiles / images
- connection to the Docker containers

Then the usage of the Iguana framework needs to be implemented. The framework needs to be setup to write the benchmark results to the Job Storage Triplestore.

To have a better control of the running jobs and the benchmarking service in general we recommend to add a small REST API. This API could be similar to the one of the HCS, that starts and stops the continuous checking. The API for the TBS can function like a switch, which indicates if a new benchmarking job will be started or not. If it is set to off, the current benchmarking job will be finished, but no new job will be started.

Lastly, after the performing of a benchmark, the cleanup of the Docker containers needs to be implemented.

4.4.6 User Management and Security

The Basilisk platform has no user management or any kind of access control implemented. Currently the REST APIs of the services allow interactions with any user. If the platform is needed to run publicly, some user management and further security measures are needed. That includes registering new users and user groups with different user rights. Some users should only be able to read benchmark results, while other should be able to create repositories and abort jobs.

Secondly, confidential information need to be kept secret. This is for example the OAuth-Key needed for accessing private GitHub repositories.

4.4.7 Frontend

The frontend introduces a new programming languages and frameworks. A short review of the current source code resulted in the following findings:

- Currently only a small web view is implemented
- Functionality to communicate with the REST APIs is missing

As explained in section 5.1 the priority for the frontend has been lowered. The focus for the thesis lies in finishing the main services and their functionality.

5 Implementation

This section describes the implementation of the concepts explained in chapter 4. Parts of the system were already implemented by other developers before this thesis.

As stated in section 4.4 some parts of the functionality are still missing and need to be implemented. This chapter documents the design and implementation of the explained shortcomings.

The following sections of the thesis differ slightly from the proposed time schedule and task list, which was originally planned in the thesis proposal. In Section 5.1 we explain why we had to alter the original thesis schedule based on the findings of the architecture review in section 4.4.

In section 5.2 we describe how the missing parts of the platform are designed and implemented. Possible problems or new insights found during this process are explained and dealt with.

5.1 Thesis Time Schedule

The time schedule of the thesis had to be altered to allow for more time designing and implementing the Basilisk platform.

Before starting this thesis it was hard to quantify how much implementation work was still needed. After the in-depth architecture review (4.4) it became clear that the implementation workload was greater than anticipated in the thesis proposal.

Therefore an alteration of the time schedule and work plan for the thesis is needed. Further we discuss the task priorities. First priority is the development of the core functionality for the platform. This means the main services are required to successfully register new versions of observed repositories, perform benchmark jobs and save the measured metrics to a Triplestore.

With a lower priority, functionalities like user management (4.4.6) will be dealt with. The least priority has the Basilisk frontend (4.3), since it is not necessarily needed to run the platform. Secondly it introduces more programming languages and frameworks to the project. Currently time schedule for the thesis can not provide enough time to fully acquire the understanding for this technology stack. Therefore the frontend will not be further implemented and deployed in the context of this thesis.

5.2 Revised Solution Design

The Basilisk platform is missing some key functionality to successfully run benchmark jobs. In this section we describe the designed solutions for the shortcomings listed in section 4.4.

5.2.1 Code Refactoring

As stated in section 4.4.1, an in-depth code refactoring was recommended. Before starting to design and implement new functionality, we performed an in-depth refactoring and restructuring of the code base. This created a clean code base on which all future implementations can be build on.

Code refactoring is the process of restructuring the source code of an application without changing its functionality[13].

5.2.2 Management of Repositories and Configurations

Section 4.4.2 already explains the problem of storing and managing repositories, the corresponding hooks and the Triplestore-configurations between the Hooks Checking Service (HCS) and the Jobs Managing Service (JMS).

Different solutions were considered for moving the functionality of merging the functionality of the two services.

In the designed solution, the management and storing of the repositories is moved into the JMS. This includes the corresponding REST endpoints and internal logic of the HCS that are needed for the management. The different repositories (GitHub and Docker Hub) are added over the REST API of the JMS.

In section 4.4.5 (Hooks Checking Service) it is listed, that the HCS is missing the REST endpoints for deleting repositories. Since the repository management is moved to the JMS, these endpoints are also added there.

The JMS communicates with the HCS over RabbitMQ message queues. Through these messages the HCS gets the needed information about the repositories it should observe. These include the URL and for GitHub repositories details like the observed branch and potentially an OAuth token for authenticating with the API.

The functionality used when a new release is found, does not need to be changed. When a new release is found, the HCS still sends a message containing the relevant information about the release to the JMS.

Figure 5.1 shows the restructured REST APIs and the adjusted messaging between HCS and JMS.

5.2.3 Restructure of Data Models in the Jobs Managing Service

In section 4.4.4 we reviewed the data model used for storing and managing the different configuration types inside the JMS. To mitigate the stated problems with the data model, we designed the database schema shown in figure 5.2.

This design takes advantage of persistence technology that is already part of the Spring framework. The models for the GitHub and Docker Hub are inherited from an abstract repository class, since the basic information like repository name and owner are needed for both repository types. The Spring framework automatically manages the different repositories and identifies them through the stored `DTYPE`.

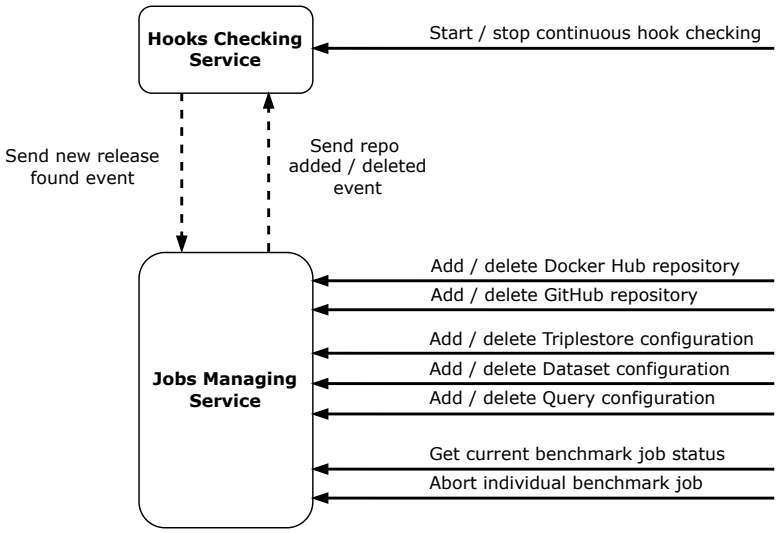

Figure 5.1: Overview of the restructured REST APIs and adjusted messaging

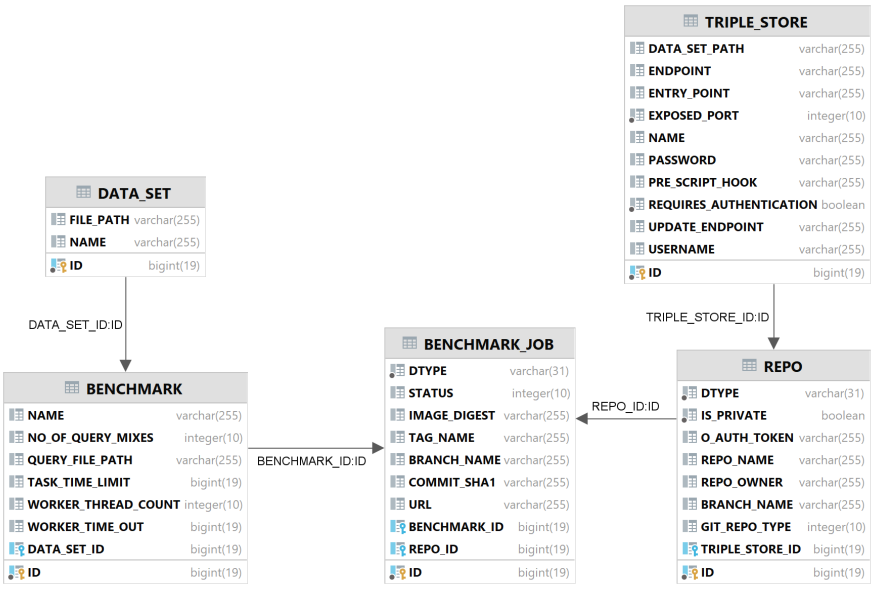

Figure 5.2: Diagram of the proposed database schema for the JMS

Secondly the relationship between Triplestore (TS) configurations and repositories is inverted. Now each repository points to a TS configuration. This means a TS configuration can be used for different observed repository setups.

Lastly the benchmark job and dataset models are cleaned up. Now each benchmark specifies a datasets and a query file as well as multiple parameters which are later used by Iguana during the execution of an benchmark job. Each benchmark is now a single entry in the benchmark table.

The individual repositories and benchmarks are then linked inside a benchmark job.

5.2.4 Creation and Management of Benchmark Jobs

Section 4.4.3 describes how the creation of benchmark jobs needed to be changed in the JMS. As explained in section 5.2.3, benchmarks are now stored in a database table and have a single dataset and query file.

When a new message about a new release arrives, the JMS will now create a benchmark job for each benchmark that is configured in Basilisk.

To manage the created benchmark jobs, new API-endpoints are created to get a list of all benchmark jobs and to abort jobs that have not yet been started.

5.2.5 Hooks for Pull Requests in GitHub Repositories

Currently the Basilisk platform can not check for new pull request published to an observed repository. This functionality would greatly support the continuous benchmarking during the development process of Triplestores.

As explained in chapter 4, Triplestores are often developed by teams who collaboratively work on Git repositories. A standard way of introducing a newly developed feature to a source code repository is a pull request. Pull requests contain a description of the proposed changes, as well as the name of the development branch which should be pulled into the main branch of the repository.

Often these changes are developed in a forked repository. A forked repository is basically an independent copy of the main repository. GitHub provides functionality to merge the latest changes of the original repo into the forked repo. To send changes from the forked repo to the original repo, a pull request is needed.

In this forked repo the developer can work independently on his changes and later create the pull request to the original repository.

The difficulty for the Basilisk platform is, that these pull requests can stem from these forked repositories. Since the repository containing the changes has a different URL to the original repo observed by the Basilisk platform, more information than usual are required to create and run a benchmark job for a pull request.

The solution we designed for this issue is an extended message. This message gets send in the situation in which the pull requests originates from a different repository. The message contains the URL, repository and branch name for the GitHub repository. Therefore the message handling in the HCS and JMS needs to be adjusted to handle this new message type.

The benchmark jobs and the Triplestore Benchmarking Service does not need to be changed. It is not relevant for a benchmark if the repository, from which the Docker container is build, is different too the observed repository.

5.2.6 Missing Implementations of the Jobs Managing Service

In this section we describe how the missing implementations are developed that are listed in section 4.4.5.

The tasks for the HCS are already dealt with in sections 5.2.2 and 5.2.5. Also the tasks for the JMS regarding the management and aborting of benchmark jobs are dealt with in section 5.2.4.

Lastly only the REST APIs for adding and removing the configurations of Triplestores and benchmarks needed to be added to the JMS. Since the basic functionality of those endpoints are similar to the endpoints for adding and removing repositoriesm, it is straight forward to implement those endpoints. The configuration and relationships of the added data models are based on the data model designed in section 5.2.3.

5.2.7 Triplestore Benchmarking Service

The implementation of the Triplestore Benchmarking Service was lacking big parts of its functionality. To explain the implementation steps, we follow the benchmarking process that is used when a new benchmark job is send to the TBS.

Waiting for jobs, getting aborted

First a Docker image needs to be created, that contains the Triplestore that will be benchmarked. Secondly a container is build and configured from the image. The configuration is stored in the data model of the JMS and is provided with the benchmark job to the TBS. Then the Iguana configuration file is created and the framework gets started. After Iguana performed the benchmark on the Container, Iguana writes the results to the Job Storage Triplestore (JSTS) and the TBS stops and removes the Docker container.

The implementation of this process is explained in the following sections.

Creation of Docker Images

The setup and creation of the Docker images containing the Triplestore for a benchmarking job can be divided into two branches. Triplestores from Docker Hub repositories are easier to setup than Triplestores stored in GitHub repositories.

For a benchmark of a Triplestore that is configured as a Docker Hubrepository, the only task is to pull the Docker image from Docker Hub by providing the repository name and owner and the image tag that marks the new release.

The creation of a Docker image from a GitHub repository is more complicated and error-prone. First the source code of the repository has to be downloaded. Then the Dockerfile needs to be located and a build needs to be initiated.

After the creation of the Docker image, the benchmark process is the same for both repository types.

Since the image creation from a GitHub repository is more complicated, we focused on the benchmark process of Docker Hub repositories. As explained in the time schedule (5.1) the first priority was to implement a fully running process, which was easier to accomplish for a Docker Hub repository.

Creating and Starting a Docker Container

After the Docker image is available, a Docker container is configured and started.

In the container the Triplestore will run that is the target of the benchmark job. For a successful benchmark the Triplestore needs to be accessible from the TBS and also needs access to the dataset to calculate the results for the benchmark queries.

To be accessible from outside the container, the container is configured to expose and listen to specific ports on the network. Through these ports the SPARQL endpoint will be accessible, which Iguana uses to send queries and receive the results.

To give the Triplestore access to the dataset of the benchmark, the dataset can be provided in two different ways. Either it is available inside the Docker container through a published volume provided by the server. Then the Triplestore can use the file directly from the file system. The second option is to provide the dataset by uploading it through the SPARQL endpoint of the running Triplestore. For this option a shell script needs to be provided by the Triplestore configuration. This script is executed by the Iguana framework before the benchmark is started.

After these configurations, the container gets started and the actual benchmark can run.

Configuration and Start of the IGUANA Framework

The actual benchmark is performed by the Iguana framework (2.3.1). The framework takes a YAML or JSON file containing the benchmark configuration as the start parameter. Therefore the TBS creates this configuration file from the provided information of the current benchmark job. This contains the name of the dataset, the address of the SPARQL endpoint and the configuration of the benchmark to be performed. The benchmark configuration contains the location of the query file and possible time limits or thread counts for the Iguana-workers. Lastly the connection for the JSTS is provided. This connection is used after the benchmark to write the benchmark results to the storage. This configuration gets encoded in JSON and is written to a temporary file on the server.

The next step is to start the benchmark by starting the Iguana framework with the created benchmark configuration. The Iguana framework is located on the server as an executable jar file. It gets started as an individual process on the server with the configuration file as the argument. If the dataset needs to be loaded after the Triplestore is started, Iguana executes the provided loading script first, before starting the actual benchmark.

Benchmark Cleanup

When the Iguana framework has finished the benchmark some cleanup is happening to free server resources. This includes removing the Iguana configuration file, the Docker container and the Docker image. The Docker container gets stopped and removed by the TBS. After that the Docker image is also removed to free up disk space.

5.3 Deployment

After the implementation phase, the Basilisk platform is deployed on a virtual server. The virtual server used is provided by the IRB (Informatik Rechnerbetrieb) of the computer science department at Paderborn University. Figure 5.3 show the specification of the VM. The VM was chosen to be powerful and to have a lot of memory and storage. Database benchmarks are often memory intensive and require a lot of storage for the different datasets.

Specifications	
CPUs	16 cores
Memory	128GB
Storage	2TB
Operating System	Debian GNU/Linux 11 (64bit)

Figure 5.3: Specification of the virtual machine on which the platform is deployed.

6

Evaluation

7

Summary and Discussion

- Summary of the work - Highlighting the key findings of the evaluation stage

Bibliography

[1] Iguana Documentation - Configuration, http://iguana-benchmark.eu/docs/3.2/usage/configuration/.

[2] Iguana Documentation - Metrics, http://iguana-benchmark.eu/docs/3.3/usage/metrics/.

[3] Microservices, https://martinfowler.com/articles/microservices.html.

[4] RabbitMQ website, https://www.rabbitmq.com/.

[5] RDF 1.1 Concepts and Abstract Syntax, https://www.w3.org/TR/rdf11-concepts/.

[6] RDF 1.1 Turtle, https://www.w3.org/TR/2014/REC-turtle-20140225/.

[7] Güneş Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. Diversified Stress Testing of RDF Data Management Systems. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig Knoblock, Denny Vrandečić, Paul Groth, Natasha Noy, Krzysztof Janowicz, and Carole Goble, editors, *The Semantic Web – ISWC 2014*, Lecture Notes in Computer Science, pages 197–212. Springer International Publishing.

[8] Anshul Bansal. DAO vs Repository Patterns | Baeldung, https://www.baeldung.com/java-dao-vs-repository.

[9] Alexander Bigerl, Felix Conrads, Charlotte Behning, Mohamed Ahmed Sherif, Muhammad Saleem, and Axel-Cyrille Ngonga Ngomo. Tentris – A Tensor-Based Triple Store. In Jeff Z. Pan, Valentina Tamma, Claudia d' Amato, Krzysztof Janowicz, Bo Fu, Axel Polleres, Oshani Seneviratne, and Lalana Kagal, editors, *The Semantic Web – ISWC 2020*, volume 12506 of *Lecture Notes in Computer Science*, pages 56–73. Springer International Publishing.

[10] Felix Conrads, Jens Lehmann, Muhammad Saleem, Mohamed Morsey, and Axel-Cyrille Ngonga Ngomo. Iguana: A Generic Framework for Benchmarking the Read-Write Performance of Triple Stores. In Claudia d' Amato, Miriam Fernandez, Valentina Tamma, Freddy Lecue, Philippe Cudré-Mauroux, Juan Sequeda, Christoph Lange, and Jeff Heflin, editors, *The Semantic Web – ISWC 2017*, volume 10588 of *Lecture Notes in Computer Science*, pages 48–65. Springer International Publishing.

[11] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow.

[12] Martin Fowler. *Patterns of Enterprise Application Architecture*. The Addison-Wesley Signature Series. Addison-Wesley.

[13] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Signature Series. Addison-Wesley, second edition edition.

[14] Martin Fowler. Repository Pattern, https://martinfowler.com/eaaCatalog/repository.html.

[15] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. 3(2):158–182.

[16] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language, https://www.w3.org/TR/sparql11-query/.

[17] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge Graphs. 54(4):71:1–71:37.

[18] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Noy, and Eva Blomqvist, editors, *The Semantic Web – ISWC 2011*, Lecture Notes in Computer Science, pages 454–469. Springer.

[19] Michael Roder, Denis Kuchelev, and Axel-Cyrille Ngonga Ngomo. HOBBIT: A platform for benchmarking Big Linked Data. page 21.

[20] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework. In Marcelo Arenas, Oscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d' Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *The Semantic Web - ISWC 2015*, Lecture Notes in Computer Science, pages 52–69. Springer International Publishing.

[21] Muhammad Saleem, Gábor Szárnyas, Felix Conrads, Syed Ahmad Chan Bukhari, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks. In *The World Wide Web Conference*, pages 1623–1633. ACM.

[22] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP2Bench: A SPARQL Performance Benchmark.