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

Bachelor's Thesis

Submitted to the DICE 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, June 8, 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. The process of benchmarking triplestores can be very time-consuming. When a new version of a triplestore is released, it has to be installed and set up to perform a benchmark on it.
The idea of the Basilisk platform is, that the process of benchmarking new triplestore releases gets fully automated. To accomplish this, the platform can observe Docker Hub and GitHub repositories. When a new release is detected, a benchmark job is created. The platform automatically downloads the new release, sets up a Docker container, and configures the IGUANA framework before starting the benchmark. After the benchmark, the results measured by IGUANA are stored in the result triplestore.
During this thesis, we analyzed and reviewed the existing implementations of the Basilisk platform. We then implemented the entire benchmark process for repositories on Docker Hub and prepared the benchmark process for GitHub repositories. Finally, we evaluated the automated benchmark process and compared it to a manual setup using the IGUANA framework.

Contents

1 Introduction

In the field of Semantic Web, knowledge graphs are the central structure to represent data and its relationships. Some specialized knowledge graph databases are required to easily store and query the data in knowledge graphs. The particular kind of database developed to store knowledge graphs is called triplestores.

Since knowledge graphs can contain vast amounts of data of various structures, which can also be subject to many changes, triplestores need to be able to handle many different workloads. Some scenarios need to handle vast amounts of data being added. In other scenarios, the triplestores need to handle many changes on the stored data or the triplestores have to deal with complex ontologies, or complex SPARQL queries. To better test and compare triplestores in these diverse scenarios, benchmarks are performed to allow an appropriate comparison between different triplestores [17].

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 [16, 14]. In the context of triplestores, a benchmark consists of creating a given knowledge graph on which multiple queries and operations are performed [6].

Usually, triplestores are developed in long iterations and are benchmarked only in a late stage of such a development iteration. Benchmarks and the evaluation of their results are often done manually and bind the time of developers. Thus, performance regressions are found very late or never.

Several benchmarks for triplestores have been proposed [17]. Most benchmarks are executed in their own execution environment. To better compare different benchmarks, benchmark-independent frameworks have been developed. IGUANA is such a benchmark-independent execution framework [6] that can be used to measure the performance of triplestores under several parallel query requests. Currently, the benchmark execution framework needs to be configured and started manually for every new benchmark configuration.

In this thesis, we continue the development of the Basilisk[1] platform and deploy an instance to a virtual machine. The platform's development was started by members of the DICE Research Group but was not completely finished.

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 continuously for

[1] `https://github.com/dice-group/Basilisk`

new versions of a given triplestores and automatically starts 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 should automatically setup a benchmark environment and start a benchmarking suite.

This means that developers do not have to worry about performing benchmarks at different stages of development. A developer has to configure a repository containing a triplestore once in the Basilisk platform. With every new release, a new benchmark will be performed without further interaction by the developer. When the developer is interested in the benchmarking results, he can send a SPARQL query to the result storage triplestore.

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 necessary for understanding this thesis. This consists of topics from the field of Semantic Web and topics from the field of software development. In chapter 4, we describe and review the architecture used in the Basilisk platform. Chapter 5 presents the development and implementations that are performed to finalize the platform. Finally, in chapter 6 we evaluate the platform and its provided benefits to the triplestore benchmark process.

2 Related Work

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

Several benchmarks have been proposed and developed to test triplestores [4, 10, 14, 16, 18]. 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. An introduction to benchmarking, in general, is given 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 [10]. These synthetic benchmarks have the offer the benefit of being able to be generated to arbitrary sizes. The main criticism for synthetic benchmarks is that the generated data can quickly become skewed and repetitive. Often the generated scenarios are criticized for not being representative enough of a real-world scenario [16]. In the following paragraphs, we introduce three benchmarks that are often mentioned when considering synthetic triplestore benchmarks.

The Lehigh University Benchmark (LUBM) Benchmark [10] 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. Fourteen extensional queries are provided that represent and test a variety of properties.

Another synthetic benchmark is SPARQL Performance Benchmark (SP²Bench) [18]. The data generated stems from the DBLP scenario. During the generation process, the generated key characteristics and word distributions are chosen to match the distributions of the original DBLP dataset. The provided queries are primarily complex, and the mean size of the result sets is above one million [16]. They also test for SPARQL features like union and optional graph patterns.

The Waterloo SPARQL Diversity Test Suite (WatDiv) generates a synthetic benchmark and consists of multiple tools [4]. The first tool is the data generator that 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. Lastly, the queries are generated with the query generator, which instantiates the templates with actual

Resource Description Framework (RDF) terms from the generated dataset. For each template, multiple queries can be generated. The benchmark only focuses on SELECT queries that do not use the union and optional pattern features of SPARQL.

2.2 Real-World Benchmarks

Real-world benchmarks use real datasets that were used in productive settings or are still being used. The actual queries are often taken from query logs of triplestores, and the datasets are based on real datasets [14, 16].

FEASIBLE is a benchmark generation framework that generates queries from provided query logs [16]. This has the advantage that the data used for the benchmark could stem from queries about a specialized real-world topic rather than an abstract synthetic model. FEASIBLE can also generate queries for the other SPARQL query types besides SELECT.

2.3 Benchmark Execution Frameworks

As the name suggests, benchmark execution frameworks 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 comparing between benchmarks difficult. Often those environments are specialized for the given benchmark and are not easily interchangeable [6].

The following sections focus on benchmark-independent execution frameworks.

2.3.1 IGUANA

Iguana is a SPARQL, benchmark-independent execution framework [6]. 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, as well as 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 allow 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 [15]. It is an extensive framework that needs to be deployed on a local cluster or online computing services like Azure[1] or AWS[2]. The platform's deployment and deploying new benchmarks to the platform can be challenging for new users of the system [15]. The data for benchmarks has to be stored in docker containers or it needs to be generated or downloaded before a benchmark, which increases the complexity of the system. The data is then sent via message queues to the benchmarked system.

With the Basilisk platform, we developed a specialized solution for continuous benchmarking of triplestores which does not need the technical complexity present in the Hobbit framework.

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

Implementing this functionality into the Hobbit framework would introduce another level of complexity to the HOBBIT system. Basilisk focuses on a smaller use-case of benchmarking SPARQL endpoints continuously with as little overhead as possible.

3 Background

This chapter introduces the fundamental topics and background information we use throughout this thesis. The Basilisk platform which is developed in this thesis focuses on benchmarking triplestores. Triplestores are used in the research field of Semantic Web, therefore we give a short introduction to the most essential topics from that research field. Since the platform is actively developed during this thesis, software development is the second big topic we introduce in this chapter.

In section 3.1, building blocks of the Semantic Web are introduced. In section 3.2, required topics from the field of software development are covered.

3.1 Semantic Web

The following topics stem from the research area of the Semantic Web. Semantic Web is the research field that tries to represent information on the internet in a way that makes it processable by computers [12].

Since this thesis focuses mostly on the implementation and deployment of the Basilisk platform, these topics are mainly 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 small scenarios. The knowledge stored in Knowledge Graphs is modeled in a graph-based structure. Nodes represent entities that are connected by various types of relations, represented by labeled edges in the graph. This has the benefit of being able to represent complex relations between different nodes and edges [13].

The most straightforward Knowledge Graph consists of three elements: The subject entity, the object entity, and the labeled edge between them describing their relation. The relation is also called the predicate. This atomic data entity consisting of subject, predicate, and object is called *triple*. In figure 3.1, a simple example of a Knowledge Graph is shown.

This Knowledge Graph could be extended in all directions. For example, the "Berlin" entity could get an edge labeled "population" which would point to Berlin's population size.

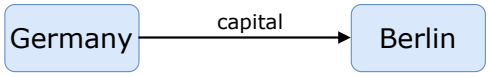

Figure 3.1: Simple Knowledge Graph

Since a graph structure is hard to store in a classic relational database, a different storage types are needed. The particular kind of database developed to store knowledge graphs is called triplestores. Triplestores will be explained in section 3.1.3.

3.1.2 RDF

The Resource Description Framework (RDF) is a framework for describing data and knowledge in a standardized way [2]. 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 Internationalized Resource Identifiers (IRIs), blank nodes or data typed literals.

RDF graphs can be encoded with different syntax styles. A popular syntax is TURTLE [3], 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 as seen in listing 3.1. The first two lines of the TURTLE document define abbreviations for the used IRIs so that the triple in line three is more readable.

Listing 3.1: RDF example in TURTLE syntax

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

3.1.3 Triplestore

Triplestores are a special kind of database developed to easily store and access knowledge graphs through queries [**?**]. They differ from relational databases by being purposefully built to only store and access data in a set of triples. Data is accessed through the query language SPARQL, which gets introduced in section 3.1.4. Often more extensive datasets can be imported or exported using RDF or another syntax.

Example of triplestores are Tentris[1], GraphDB[2], Virtuoso[3], or Jena TDB[4]. Since Tentris is actively developed by our research group, we will focus our tests and evaluations of the platform on this triplestore.

This thesis focuses on triplestores that has a SPARQL endpoint 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) [11] is a query language for manipulating and retrieving RDF data stored in triplestores. Like RDF, SPARQL is part of the W3C recommendations for technologies in the semantic web.

[1] https://tentris.dice-research.org/
[2] https://graphdb.ontotext.com/
[3] https://virtuoso.openlinksw.com/
[4] https://jena.apache.org/documentation/tdb/

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, followed by a `WHERE` clause giving restrictions and conditions.

Queries can contain optional graph patterns, conjunctions, disjunctions, and aggregation functions. These extensions can help formulate more complex queries.

There are two example SPARQL queries in listings 3.2 and 3.3. Executed against the DBPedia SPARQL endpoint[5], the following results can be found: The first example query (3.2) requests the variable which matches the `WHERE` clause searching for the capital of Germany, which is `dbr:Berlin`. The second query (3.3) 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.

Listing 3.2: SPARQL query searching the capital of Germany

```
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 }
```

Listing 3.3: SPARQL query searching all relations between Germany and Berlin

```
1 SELECT ?relation
2 WHERE {
3         dbr:Germany ?relation dbr:Berlin .
4 }
```

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 (3.2.2) and Microservice Architecture (3.2.3) explain the basic idea and concept of the microservice architecture. In the sections RabbitMQ and Spring (3.2.5, 3.2.4), 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 workload in 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., the number of executed queries (NoQ), queries per second (QPS), and query mixes per hour (QMPH) [1].

A distinction is made between micro and macro benchmarks [9]. 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 cover macro benchmarks.

[5] `https://dbpedia.org/sparql`

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 [7]. 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.

A typical example is an online shop system that is divided into microservices. One service of this system could be managing the customer data, like contact details and shipping addresses. If another service needs this information, it can send a request to this service over defined protocols.

3.2.3 Microservice Architecture

Microservice architecture is a way of designing a software application as a set of microservices that interact with each other to provide the designed functionality [7, **?**]. The application's functionality gets split up into microservices that interact only through a defined message protocol. This allows for a distributed system in which the individual services could be implemented in different programming languages and also could be located on different servers.

Extending the example from section 3.2.2 we have a microservice architecture with three services. One service manages the customer data, a second service manages the incoming orders, and the last service manages the shipping process. When an order arrives in the order service it will send a message to the shipping service, with the order items. The shipping service will then request the customer's shipping address from the customer-data service.

3.2.4 Spring and Spring Boot

Spring[6] is a widespread open-source Java framework that facilitates the development process for various kinds of java applications and systems.

Spring Boot[7] is an extension to the Spring framework that follows the convention-over-configuration design paradigm. This means that the implementation of applications has to follow standard design conventions that replace the need for configuration files for many standard scenarios. Spring Boot also comes with pre-configured standard libraries for the Spring platform to ease the development of many standard applications like web apps or microservices.

Spring and Spring Boot use different annotations to decorate classes and methods. These annotations configure the classes 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.

3.2.5 RabbitMQ

RabbitMQ[8] 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 and flexible routing.

Since RabbitMQ is a widely-used message broker, the Spring framework (3.2.4) already comes with the needed libraries to work with the RabbitMQ system.

[6] `https://spring.io/`
[7] `https://spring.io/projects/spring-boot`
[8] `https://www.rabbitmq.com/`

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. RabbitMQ ensures reliable delivery of messages by using acknowledgments and confirms of the consumers[9]. If a message is lost in transit or the consumer encounters an exception, the message is not deleted from the queue. Only if the consumer sends a confirmation to the RabbitMQ broker the message is deleted from the queue. The RabbitMQ broker also ensures the durability of the queues and messages through file backups.

3.2.6 Docker

Docker[10] is a platform for containerizing applications for development and deployment. Following the idea of a shipping container used on freight, a Docker container can be installed on any system that supports the Docker technology. That means that there are no other software requirements than the Docker engine to run a container. Inside the container, all requirements and libraries are installed that are needed by the application that is run in the container.

The application can only reach the outside of the container and, vice-versa, can be reached from the outside, if the required ports are published. This encapsulates the application from unwanted accesses, and protects the host system from malicious software that could be run inside a container.

A Docker container is always built from a Docker image. A Docker image is a read-only template to create a Docker container with the running application inside.

There are two main ways to create a Docker image. First, it could be downloaded from Docker Hub. The second option is to build the Docker image from a Dockerfile. A Dockerfile[11] contains the instructions for building a Docker image. The Dockerfile first references the base image, which is used to build the image. This could be, for example, a standard Ubuntu distribution. After the base image, the Dockerfile defines which commands are run or which files should be copied from the host system into the image.

After the image is built or downloaded from Docker Hub, the container can be further configured before it gets started. For example, ports can be published to make the service running inside the container available to the outside.

Often multiple containers are needed to deploy a complete service platform. For example, the Basilisk platform consists of three containerized services, a RabbitMQ container, and a container running Fuseki. For these situations, Docker offers Docker Compose[12] files, which help to orchestrate multiple containers in a Docker network.

[9] `https://www.rabbitmq.com/reliability.html`
[10] `https://docs.docker.com/get-started/overview/`
[11] `https://docs.docker.com/engine/reference/builder/`
[12] `https://docs.docker.com/compose/`

4 Approach

In this chapter we give an overview to the design of the Basilisk platform. We will explain the different processes used in the platform in section 4.2. In section 4.4 we will analyze and review the current software architecture and implementation status of the Basilisk platform.

– Welche fragen was ist die intenzion, problem erklären warum sinnvoll permament warum neue benchmarks immer quali überprüfen – brücke approach was gemacht dann detail was sind benchmarks, ts, benchmarks auf ts

mehr roter Faden, Verbindung zw den genannten Punkten

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 is that the Basilisk platform will automatically check for a new release of a registered triplestore repository and will then perform benchmarks on this release.

überleitung warum benchmarks relevant..

Benchmarks are also relevant during the development process. A benchmark performed automatically, for example, when a new pull request is added, 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 can be run on different hardware systems and they interact with each other via the RabbitMQ (3.2.5) 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 2.6.6.

The package structure used for implementing the services is similar in all three services. It is strongly influenced by the structure recommended for the Spring Boot framework.

4.2 Main Services

The next sections explain the idea of the three main services of the Basilisk platform, namely Hooks Checking Service (HCS) (section 4.2.1), Job Managing Service (JMS) (section 4.2.2), and Triplestore Benchmarking Service (TBS) (section 4.2.3).

This explanation follows the flow of actions that happen while a user configures a continuous benchmark and the process that happens when a benchmark is initiated. This is based on code review and analysis and information and diagrams 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 essential messages sent between the services and the interactions with GitHub and Docker Hub. Messages sent between the services are sent using a RabbitMQ message broker. The users interact with the services through REST endpoints which are not shown in this figure. Interactions of the users with the REST APIs are stateless HTTP requests. The services and the messages will be explained in the following sections.

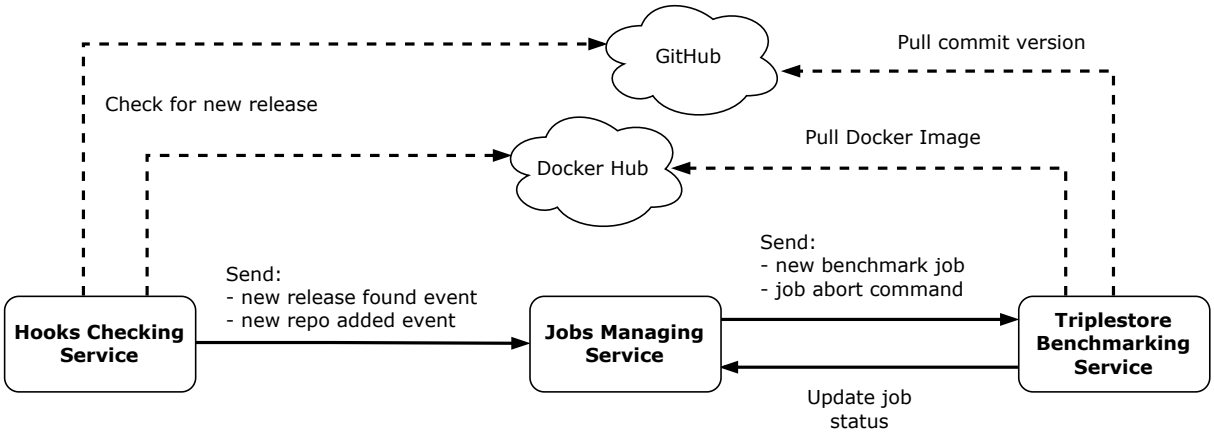

Figure 4.1: Overview of the three microservices

4.2.1 Hooks Checking Service

The main task of the 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, a triplestore is registered at the HCS by defining the repository (GitHub or Docker Hub) that has to be observed for changes. This happens through REST API calls to the HCS providing the repository's name and owner. The HCS will then create a hook for the repository to get noticed about changes. In general, a hook is 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 a

repository. The service will query the saved repositories regularly by polling the REST APIs of GitHub and Docker Hub 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 Job 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 endpoints are for adding and deleting GitHub and Docker Hub repositories. Figure 4.2 shows these endpoints.

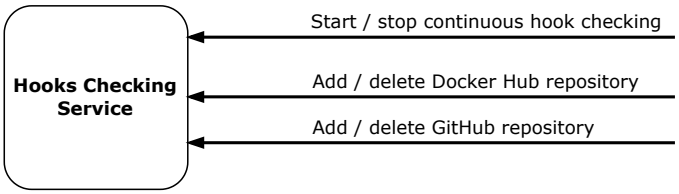

Figure 4.2: REST API of the Hooks Checking Service

The communication between the HCS and the Job Managing Service is done with RabbitMQ (3.2.5) messages, by using the Job Request Queues. The messages contain different events that can occur in the HCS. For example, an event is sent when adding or deleting a repository, or a new release is detected.

4.2.2 Job Managing Service

The Job Managing Service (JMS) has two core functions: first, it manages configurations of triplestores. These are used for their second function, creating and managing benchmarks jobs. Lastly, the JMS manages the status for running and pending jobs sent to the TBS.

There are three configuration types needed for a benchmark job. First, the platform needs the configuration for the triplestore. This configuration[1] includes, for example, the SPARQL endpoint as well as the user and password for the connection to the endpoint. The Iguana framework needs these arguments to properly connect to the triplestore under test.

Secondly, the platform needs configurations for datasets and query files. The dataset configuration consists of the dataset name and the URL for the dataset's location on the server. The query configuration consists similarly of a name for the queries and the URL for the location of the query file.

All of the named configurations are managed over the REST API of the JMS.

When the JMS receives an event from the HCS 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 are created. Each benchmark job has a query file and a dataset that is used for the benchmark.

These benchmark jobs will then be sent to the TBS over the Benchmark Job Queue.

[1] `http://iguana-benchmark.eu/docs/3.2/usage/configuration/`

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

API and Messaging

The JMS communicates with the HCS and the TBS over RabbitMQ message queues. Repository events are received from the HCS and benchmark job events are send to the TBS over the Benchmark Job Queue (BJQ).

Interaction with the user is handled over the REST API. The API offers endpoints for adding and deleting the different configurations of triplestores, datasets, and queries. The 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.

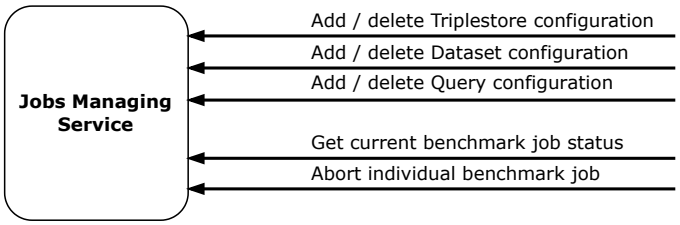

Figure 4.3: REST API of the Job Managing Service

4.2.3 Triplestore Benchmarking Service

The Triplestore Benchmarking Service (TBS) executes the benchmark jobs it receives from the JMS and saves the benchmarking results to the Job Storage Triplestore.

To execute a benchmark, the service needs a running instance of the triplestore with the loaded dataset on which the benchmark will be executed. This instance is built 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. The service 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. The service then runs the image as a Docker Container.

Loading the dataset depends on the triplestore. Some triplestores like Tentris[2] can load a dataset on startup. Other triplestores like Oxigraph[3] require the dataset to be loaded through a REST endpoint after the triplestore is started.

After starting the Docker Container, the TBS creates a configuration file for the Iguana framework. Iguana will then perform the benchmark with the provided configurations.

When the benchmark is finished, the results are written to the Job Storage Triplestore (JSTS).

[2] `https://tentris.dice-research.org/`
[3] `https://github.com/oxigraph/oxigraph`

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 set up new repositories, triplestores and datasets. Further, the user can request information about current benchmark jobs, abort or remove pending jobs. Lastly, the frontend can request and visualize the benchmarking results stored in the Job Storage Triplestore.

4.4 Architecture and Code Review

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

4.4.1 Code Refactoring

Code refactoring is the process of restructuring the source code of an application without changing its functionality [8]. Some inconsistencies in the code style and duplicate code snippets have been found during the code analysis. In other parts, the code structure differed from the design patterns recommended for the Spring and Spring Boot framework.

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

4.4.2 Management of Repositories and Configurations

Previously, the observed repositories were managed and stored in the HCS while the configurations for the triplestores were managed and stored in the JMS. This made it difficult to internally link a repository to a triplestore configuration since they were stored in different services.

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

Therefore we recommended restructuring the management of repositories.

4.4.3 Creation and Management of Benchmarking Jobs

When a new release is found by the HCS, the Job Managing Service will create and manage benchmarking jobs which will be executed by the TBS. Previously, the JMS had created multiple

jobs. For each query file, a job was created for each dataset. This means that each dataset was mixed with each query file, which led 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 had to be changed, and 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 previous database schema.

The schema had logical errors and was, in parts, incomplete. In the following, we list some inconsistencies and possible problems that we noticed:

- The only way to identify a repository as GitHub or Docker Hub repository was to check in the triplestore configuration. If a repository was assigned to the wrong type, the resulting benchmark job was not executable because GitHub and Docker Hub need to be handled differently during a benchmark, as explained in section 4.2.3.

- Each triplestore configuration could have exactly one GitHub or Docker Hub repository. This means that for every repository, a new triplestore configuration had to be added. There are fewer duplicate configurations if multiple repositories point to the same configuration. For example, a hook is set up to observe a GitHub repository for new releases, and another hook is set up to observe the same GitHub repo for pull requests. In this case, both hooks are using the same triplestore configuration since it is the same triplestore that gets benchmarked.

- As explained in section 4.4.3, the creation and storage of benchmark jobs had to be changed. Previously, the data model structure for benchmark jobs, datasets, and query files were too complicated. Since the creation process of the jobs had to be changed, the data model was also changed, and the model relationships were cleaned up.

Therefore, the data model for the JMS had to be restructured to cover real world requirements better.

4.4.5 Missing Implementations

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

Hooks Checking Service

The implementation of the Hooks Checking Service was well developed. Small additions had to be implemented.

- The REST endpoints for deleting GitHub and Docker Hub repositories had to be added.

- Previously Pull Requests for GitHub repositories could not be observed.

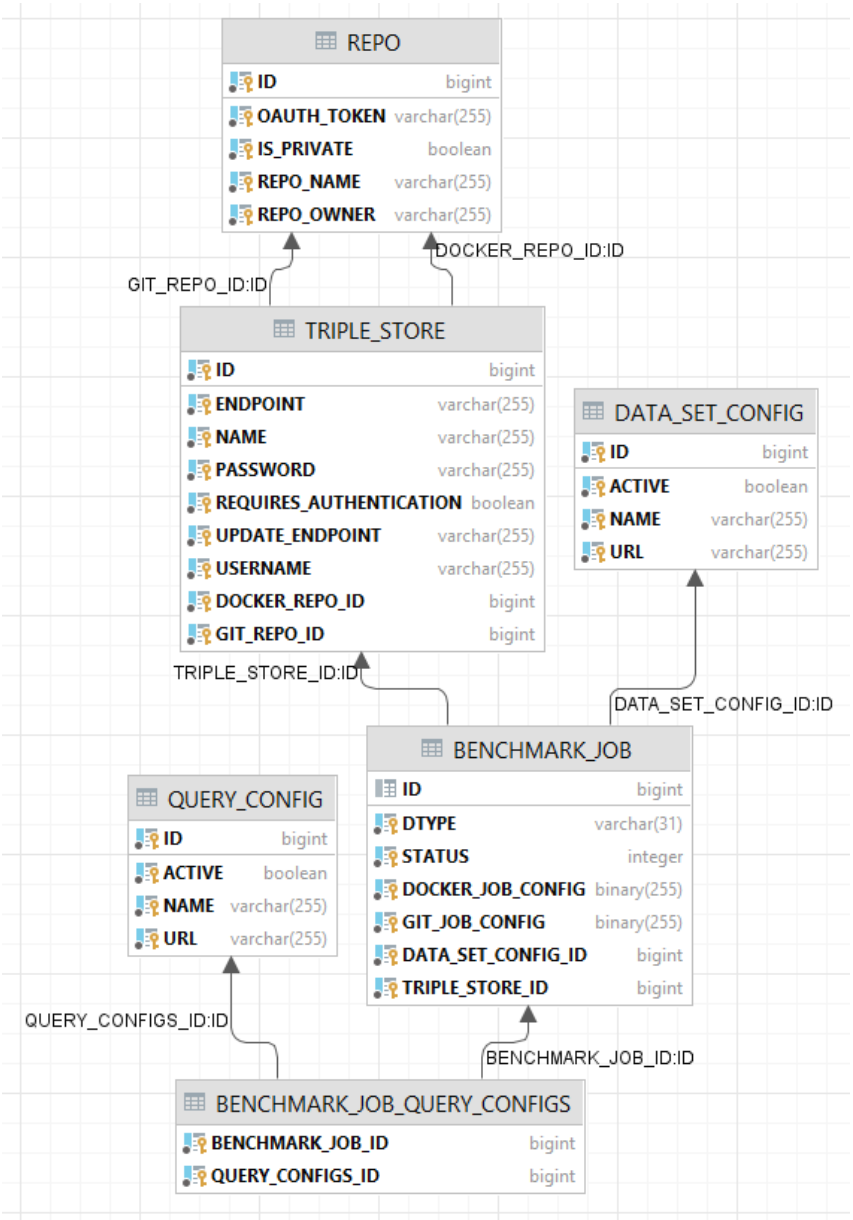

Figure 4.4: Diagram of the previous database schema used in the JMS

Job Managing Service

The implementation of the Job Managing Service was mainly missing the REST API and some internal logic. The following REST endpoints had 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 had 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 previously contained only a few classes for the data models and simple structures of service classes. Significant parts of the logic had still to be implemented.

Existing classes were mainly for storing and manipulating data models, configurations, and basic message queue interactions. These classes did not carry much functionality.

The main functionality of the TBS had to be implemented. This consisted 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 / Docker Images
- Connecting to the Docker containers

Further, the usage of the Iguana framework had to be implemented. The framework had to be set up to write the benchmark results to the Job Storage Triplestore.

To have better control of the running jobs and the benchmarking service in general, we recommended adding a small REST API to the TBS. This API is similar to the one of the HCS, which starts and stops the continuous checking. The API for the TBS functions 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 performing a benchmark, the cleanup of the Docker containers had to be implemented.

4.4.6 User Management and Security

The Basilisk platform has no user management and 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 users should be able to create repositories and abort jobs.

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

4.4.7 Frontend

The frontend introduces 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 priority of the thesis lies in finishing the main services and their functionality.

5 Implementation

This chapter 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, parts of the functionality were still missing and needed 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 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. Lastly, we explain how the platform is deployed in section 5.3.

5.1 Thesis Time Schedule

The time schedule of the thesis had to be altered to allow for more time to design and implement 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 more significant than anticipated in the thesis proposal. Therefore, an alteration of the time schedule and work plan for the thesis was needed. Further, we discuss task priorities. First, priority had the development of the core functionality for the platform. We wanted to be able to perform the entire process for at least one repository type. This means that the main services must successfully check for new versions of observed repositories, and create benchmark jobs for theses releases. Then the platform has to perform these jobs on the triplestores and save the measured result metrics to a triplestore.

Functionalities like user management (4.4.6) were marked with a lower priority and were not developed in this thesis. The least priority had 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. The time schedule for the thesis can not provide enough time to acquire the understanding for this technology stack fully. Therefore, the frontend will not be further implemented and deployed in the context of this thesis.

In the end, one more functionality had to be left unimplemented. This is performing of benchmark jobs on GitHub repositories. Although the platform can observe various configurations of GitHub repositories and can create benchmark jobs for such, the benchmark process for

these repositories introduced a lot more complexity that we were not able to implement during this thesis.

As stated above, the priority was to implement the entire Basilisk process for Docker Hub repositories, which we were able to do.

5.2 Revised Solution Design

The Basilisk platform is missing some key functionality to run benchmarking jobs successfully. 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 designing and implementing new functionality, we performed an in-depth refactoring and restructuring of the code base. This resulted in a clean code base on which all future implementations can be built.

5.2.2 Management of Repositories and Configurations

In section 4.4.2 we explained the problem of storing and managing repositories, the corresponding hooks and the triplestore-configurations between the Hooks Checking Service (HCS) and the Job Managing Service (JMS).

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

In the designed solution, the management and storing of the repositories are 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 Job 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 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

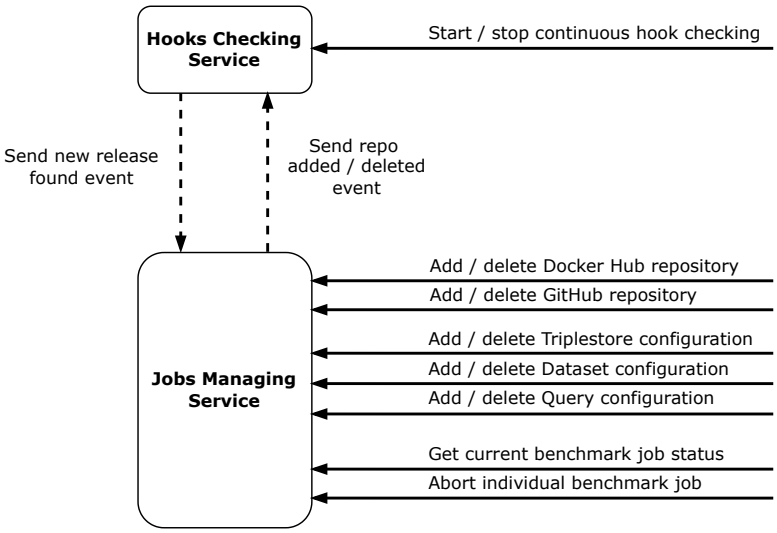

Figure 5.1: Overview of the restructured REST APIs and adjusted messaging

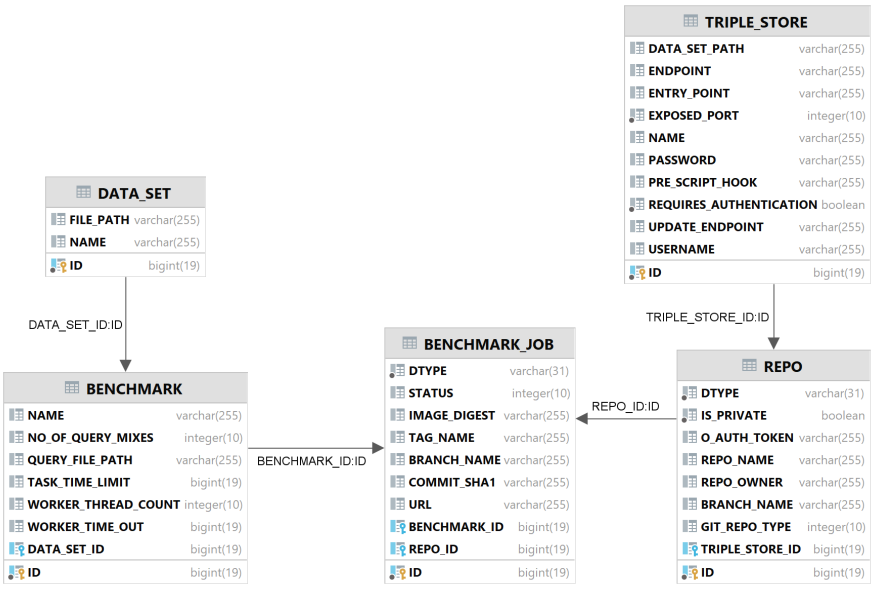

Figure 5.2: Diagram of the proposed database schema for the JMS

Spring framework automatically manages the different repositories and identifies them through the stored `DTYPE`.

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 dataset and a query file as well as multiple parameters, which are later used by Iguana during the execution of a 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 described 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 requests 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 pull request is a standard way of introducing a newly developed feature to a source code repository. Pull requests contain a description of the proposed changes, and 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 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 than the original repo observed by the Basilisk platform, more information than usual is 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 sent in situations in which the pull requests originate 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 did not need to be changed. It is not relevant for a benchmark if the repository from which the Docker container is built differs from the observed repository.

5.2.6 Missing Implementations in the Job 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 need to be added to the JMS. Since the basic functionality of those endpoints is similar to the endpoints for adding and removing repositories, it is straightforward 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 significant parts of its functionality. To explain the implementation steps, we follow the benchmarking process that is used when a new benchmark job is sent to the TBS.

The JMS sends the created benchmark jobs via the message queues of RabbitMQ. The TBS stores these jobs internally in a job queue. Benchmark jobs can be manually aborted by the user over the REST API of the JMS. When a benchmark job is aborted, the JMS will send a message to the TBS. If the job has not been processed yet, the TBS will skip the job when looking for the next job to run.

When a benchmark job is started, the TBS needs to create a Docker image first that contains the triplestore that will be benchmarked. Afterward, a container is built 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 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 Hub repository, 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 than simply downloading an image from Docker Hub. 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, which often requires additional parameters and configurations. Because of this increased complexity, we focused on the benchmark process of Docker Hub repositories. As explained in the time schedule (5.1), the priority was to implement a fully running process, which was easier to accomplish for a Docker Hub repository.

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

Creating and Starting a Docker Container

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

The triplestore which is the target of the benchmark job, will be run in the Docker container. 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. In this case, 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. The Iguana 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.

REST Endpoint to Control Job Starting

To have a better control of the execution of the benchmark jobs, the TBS is extended with a REST endpoint that can start or stop the execution service. This functionality works similarly to the HCS. When the service is started over the endpoint, the queue of benchmark jobs is read, and the next job in the queue is started. When the user decides to stop the benchmark execution service, the running job will be finished, but no new job will be started from the job queue.

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 shows the specification of the VM. The VM was configured to have sufficient resources for small- to medium-sized benchmarks.

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.

The Basilisk platform consists of the three main services described in section 4.2. For the communication between these services, the platform also needs a RabbitMQ message server. Also, a triplestore is required as the Job Storage Triplestore for storing the benchmark results.

For the platform to be easily manageable, we decided to deploy the whole platform in Docker containers. This has the advantage of using a single Docker Compose file for configuring the services and the network.

The individual services are compiled into jar files, and the Docker build copies them into individual Docker images. The build of the containers for the HCS and JMS is simple and straightforward. The container for the TBS needs two additional configurations for the service to run. First, the Iguana framework is installed inside the container. Afterward, the container is configured in the Docker Compose file to have access to the Docker socket of the server. This is needed for the service to start and manage the Docker images and containers of the triplestores that are getting benchmarked.

The RabbitMQ message server is available as an official Docker container on Docker Hub. For the JSTS we decided to use the Fuseki[1] triplestore, which also is available as a Docker container.

In total, the Docker Compose configuration consists of 5 containers that can be started and stopped with simple commands. During a benchmark, a sixth container is started, which is running the triplestore. An overview of the setup is shown in figure 5.4.

The three microservices of the Basilisk platform are available through the ports 8080 (HCS), 8081 (JMS), and 8082 (TBS) of the host server. RabbitMQ container is available through port 5672, and the Fuseki triplestore which is used as the Job Storage for the benchmark results, is reachable at port 3030 of the host server. During a benchmark, the tested triplestore is started in its own Docker container, which also has port 81 published for reaching the SPARQL endpoint.

[1] `https://jena.apache.org/documentation/fuseki2/fuseki-main`

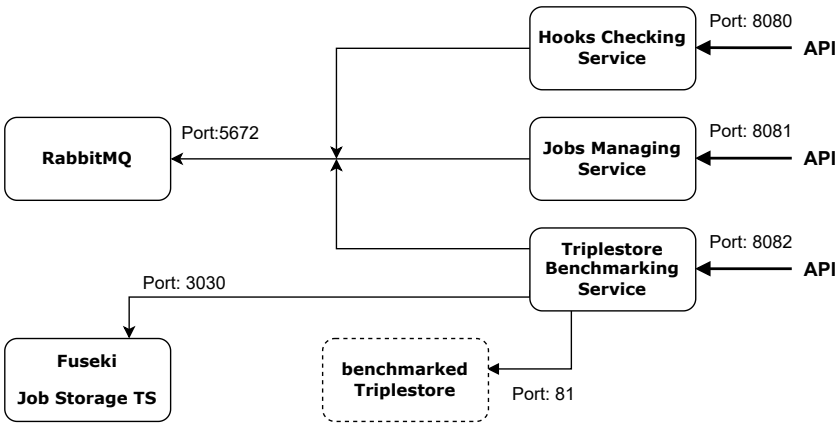

Figure 5.4: Overview of the deployed Docker containers and used Ports.

6 Evaluation

This chapter evaluates the Basilisk platform based on the developments described in chapter 5 and the ease of using the platform to set up a continuous benchmark job for an existing repository. We will compare the manual benchmarking process using IGUANA to the use of the Basilisk platform and evaluate the added value the platform creates for the benchmarking process.

The main goal of the platform is to simplify the process of benchmarking known triplestores. The platform automates the detection of a new release of a configured triplestore and automates the execution of a benchmark job for the new releases.

To evaluate the capabilities of the platform, we set up continuous benchmarks for two different triplestores. The first triplestore is TENTRIS[1], which is developed by the DICE-research group. The second triplestore benchmarked in this thesis is Oxigraph[2].

The triplestores are chosen because they both are available as a ready-to-run docker image on Docker Hub. They also differ in the way the benchmark dataset is loaded into their internal storage. The TENTRIS triplestore accepts the dataset file as an argument at program start, while Oxigraph needs to be started before the data is upoaded through the SPARQL endpoint of the triplestore.

6.1 Initial Benchmark Setup

In this section, we compare the steps needed for setting up an initial triplestore benchmark using the IGUANA framework. In general, the execution of a benchmark has the following four requirements: A running triplestore, the IGUANA framework, a dataset file, and a query file.

The following two sections describe the initial setup for a manual benchmark run and the recommended process to create a Basilisk configuration for a triplestore.

6.1.1 Manual Benchmark Setup

To manually run a benchmark, first, the triplestore needs to be installed and started. This can be done for a manual test run as a full installation or by using a Docker container. Often it is easier to use a ready-to-run docker container that contains all needed dependencies and a

[1] `https://tentris.dice-research.org/`
[2] `https://github.com/oxigraph/oxigraph`

running installation of the triplestore. On the host system, only the Docker engine is needed to run a container.

When the triplestore is running, the dataset needs to be loaded into the triplestore. This can be done either by providing the dataset during startup or by uploading the data through the SPARQL endpoint. Lastly, the Iguana framework needs to be configured by providing a configuration file containing the query file and SPARQL endpoint.

This process is similar for Tentris and Oxigraph. The only difference is in the upload of the dataset.

6.1.2 Basilisk Benchmark Setup

When a triplestore is fully configured in the Basilisk platform, the platform will automatically provide all four requirements for a benchmark when a new benchmark job is automatically created. An instance of the triplestore is started, the Iguana framework is configured, and the dataset- and query files are loaded.

To create a working triplestore configuration for the platform, we recommend developing and testing a local setup first. The process of creating this initial test setup is similar to the setup of a manual benchmark explained in section 6.1.1. However in this case, we need to use a Docker container since Basilisk is only working with a container setup.

The local setup should consist of a triplestore running in a Docker container, which is also reachable over the SPARQL endpoint. To make sure that Iguana is able to perform a benchmark, it is also advised to start a short benchmark with a simple Iguana configuration.

The Iguana configurations for the tested triplestores will slightly differ for loading the dataset into the triplestore. In case of Tentris, the dataset is configured to be provided inside the Docker container to be loaded on startup. For Oxigraph, the dataset does not need to be provided inside the Docker container. In this situation, the dataset needs to be loaded after the startup. Iguana needs to be configured with a pre-hook script that will be executed before the real benchmark starts. The task of the pre-hook-script is to take a dataset file as input and upload the file to the running Oxigraph instance. This script should be implemented and tested with the local test setup.

When a working setup is found, the setup can be transferred into the Basilisk platform. Again, the setup for Tentris and Oxigraph are mostly the same. For Oxigraph, the custom load script is provided, and the Basilisk configuration will point to the script when creating an Iguana configuration.

6.1.3 Comparison of Initial Setups

The initial setup to perform one benchmark for one triplestore version is nearly the same for the manual process as well as for the Basilisk process. In both scenarios the triplestore and Iguana are setup and run manually. The configuration of the Basilisk platform is more complicated for the case of loading the dataset through the SPARQL endpoint since a custom load script is needed. Additionally, the configuration needs to be transferred to the Basilisk platform before a benchmark can be started.

6.2 Setup of further Benchmarks

The real advantage of the Basilisk platform can be seen when further benchmarks have to be run for an already configured triplestore.

We look into two scenarios that require the run of further benchmarks on a known triplestore. Both scenarios will be looked at for the manual setup and an already configured Basilisk setup.

The first scenario is the usage of a different dataset and query file as a new benchmark that is to be run. The second scenario is the benchmark of a different version of a configured triplestore.

Both scenarios are evaluated in the following sections.

6.2.1 Using a different Benchmark

In the scenario of using a different benchmark, a new dataset and query file are used. For the manual setup described in section 6.1.1 multiple steps have to be done to update the dataset and query file. First, the dataset needs to be loaded into the triplestore. This can be done by using the SPARQL endpoint to upload the data or by restarting the triplestore and providing the new dataset at startup. For Tentris it is usually easier to restart the triplestore with the new dataset as a argument on startup. Secondly, the Iguana configuration needs to be adjusted to use the new query file.

In case of the Basilisk setup, only the new dataset and query file has to be configured in the platform. When a new benchmark job is executed, the Iguana configuration is automatically generated using the new benchmark setup. If the load-script for Oxigraph is set up correctly, the new dataset will also be automatically uploaded to the triplestore. To perform the new benchmark, a manual job can be started by sending a request to the API of the JMS.

6.2.2 Benchmarking a new Triplestore Version

If a new version of a triplestore should be benchmarked, again, multiple steps are needed for the manual benchmark process. The first step is to download the new version and start as a Docker container. Then, the dataset needs to be loaded into the triplestore, and lastly, the Iguana configuration needs to be updated to the new SPARQL endpoint location.

The Basilisk configuration does not need to be changed. If a new version has to be benchmarked, either the platform has already noticed the new version on its own and created a new benchmark job automatically, or the user can create a manual benchmark job by providing the benchmark that should be used and the triplestore version to the API of the JMS. The platform will then automatically set up the container and configure Iguana to run the benchmark job. This is the main idea why the platform was originally developed. Of course, this will only work if the other versions of the triplestore can use the same basic configuration for the startup, loading the dataset, and providing the SPARQL endpoint. If there are significant changes to the setup and structure of a triplestore, a new configuration in Basilisk is needed.

6.2.3 Comparison of Benchmark Changes

As seen in the description of the above scenarios, the triplestore configuration of the Basilisk platform is not changed at all. Only the new benchmark files are registered in case a new benchmark should be performed.

In contrast to this, the manual setup requires a lot of manual changes. Each change to a running configuration requires the user to set up the benchmark configuration from the start. Either the triplestore is downloaded and set up again, or the Iguana configuration needs manual changes. This results in a lot of manual work that is often similar to tasks that have been done for other setups.

6.3 Basilisk Evaluation

The idea of the Basilisk platform is focused on the use-case of automatically benchmarking triplestores through their SPARQL endpoints. The Basilisk platform fulfills this task of automating the benchmark process of configured triplestore repositories stored on Docker Hub.

The comparison of the Basilisk platform to the manual benchmark process shows the advantage of using Basilisk to perform multiple benchmarks on different triplestore versions.

Setting up the Basilisk platform is initially more complicated and a little more restricted than a manual setup of a triplestore. However as soon as the second benchmark is performed, the Basilisk platform has no manual setup time by the user, and the whole process is already faster than a manual approach. Once the triplestore is configured, either the benchmark jobs are created automatically by the system, or the user can manually create a job. The setup and execution of the benchmark job are then done completely automatically by the platform, and no further user interaction is needed.

Using the system, we were able to perform 16 benchmarks on different Tentris versions using the Semantic Web Dog Food (SWDF) dataset and query file in just one hour. It would not be possible to benchmark all versions manually by downloading and setting up the Docker images in a similar time frame.

The SWDF dataset contains 372K triples with 32K subjects, 96K objects, and 185 predicates. We were using the same version that was also used in [5].

Figure 6.1 shows the number of query mixes that are executed per hour on the different triplestore versions. The average of the penalized queries per second is shown in figure 6.2. A query gets penalized if it failed, e. g. if the timeout is hit, or the triplestore returned a wrong return code. Oxigraph seems to have a low QMPH while still having a high average QPS. We investigated this further in figure 6.3 which shows a boxplot of the QPS measurements taken for each query during a benchmark. For Oxigraph, we can see that some queries have a very low QPS value. This explains the low QMPH values we see in figure 6.1. The averaged QPS value gets not influenced that much by singular low QPS values, as seen in figure 6.2.

All benchmark results are available through the SPARQL endpoint of the Fuseki triplestore in which the results are written after each benchmark.

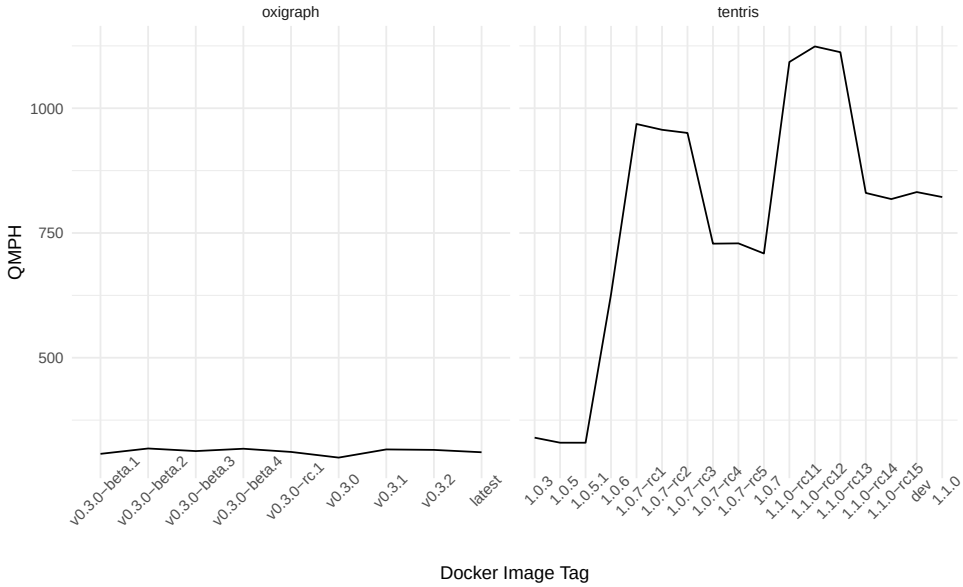

Figure 6.1: Measured QMPH of the benchmarked triplestore versions

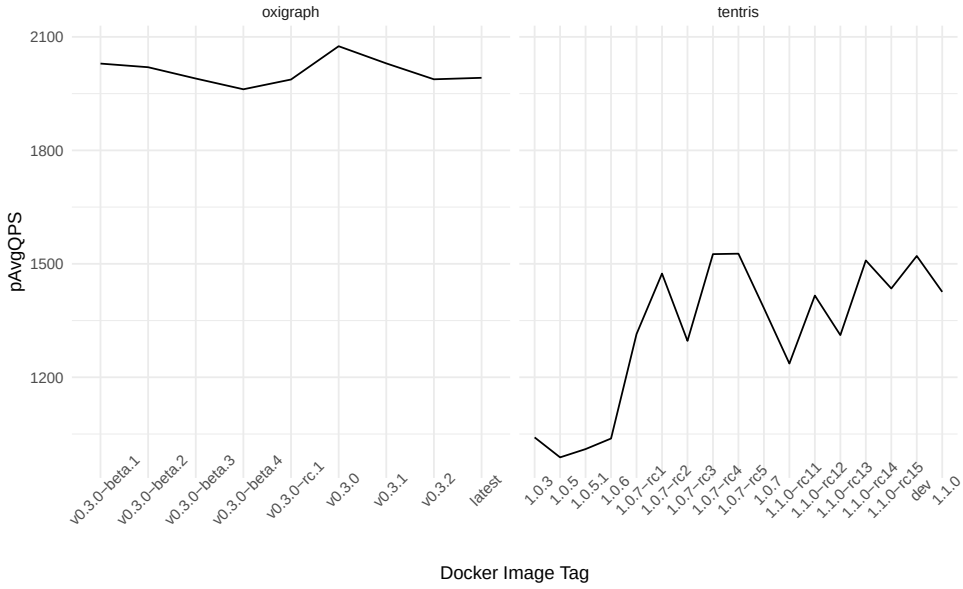

Figure 6.2: Measured pAvgQPS of the benchmarked triplestore versions

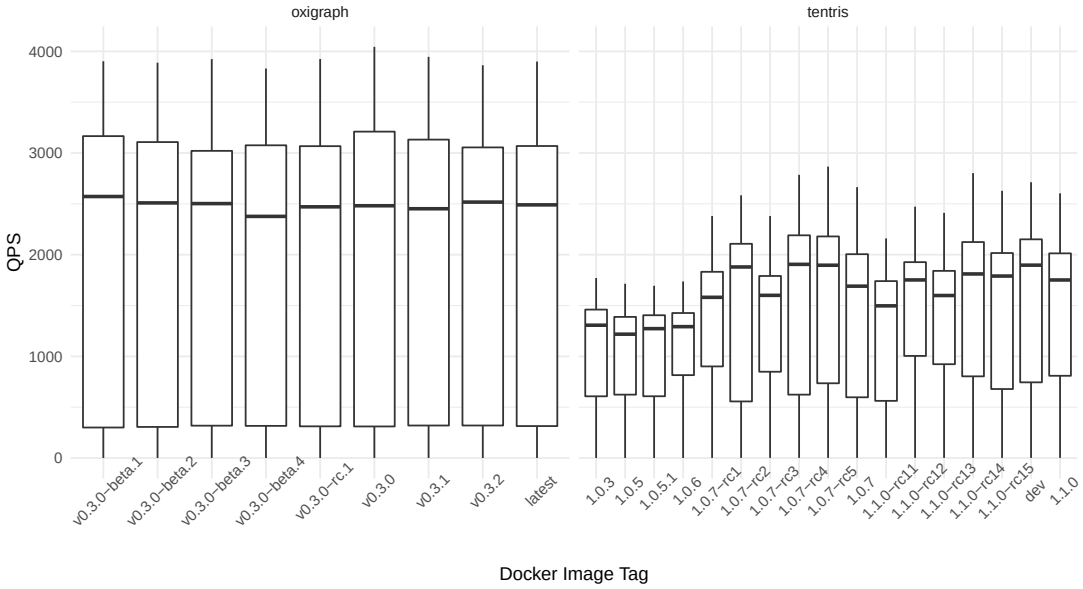

Figure 6.3: Boxplot of the QPS of executed queries for each benchmarked triplestore version

7 Summary and Future Work

In this chapter, we summarize our development of the Basilisk platform and highlight the key findings of the evaluation phase. Lastly, we point out the future work that can be done to extend the functionality of the platform.

7.1 Summary

Benchmarking of triplestores is, in general, very time-consuming and often requires a lot of configuration and setup time. Because of this, triplestores are often benchmarked late in the development process. To improve this time-consuming process and to try to automate the benchmark process as much as possible, the development of the Basilisk platform was started. The main task of the platform is defined to benchmark triplestores continuously. This means that the platform is configured once and will then automatically perform benchmarks on new versions of the configured triplestores.

In this thesis, we have continued the development of the Basilisk platform. The platform is designed to check for new triplestore releases continuously, and to automatically perform benchmarks if a new release is found. triplestore releases could be found either in Docker Hub or GitHub repositories. The complete process of checking for new releases and benchmarking new releases is fully implemented for Docker Hub repositories.

During this thesis, we have performed an in-depth architecture and code review of the platform. Based on that review, we defined implementation tasks for improving and extending the existing code base.

The main improvements were the restructuring of the microservices, and a restructure of the used data models. These changes were implemented before starting on the missing implementations of the benchmark service.

Most of the functionalities of the Triplestore Benchmarking Service was not yet implemented. The implemented functionality initializes the triplestore inside a Docker container and performs the benchmarks. During this thesis, we focused on completing the benchmark process for triplestore releases in Docker Hub repositories. To provide the same functionality for GitHub repositories, some further steps are needed, which are explained in section 7.2.

To perform the benchmarks, the benchmark-independent IGUANA framework is used. The configuration for IGUANA is generated from the information stored in the triplestore configurations inside the Basilisk platform.

Finally, the Basilisk platform was deployed on a VM hosted inside the network of the Paderborn University. On this deployment, the Tentris and Oxigraph triplestores were configured, and multiple versions of both triplestores were successfully benchmarked using the SWDF benchmark. The user only needs to start the system and configure the triplestores. No further interactions are needed during the benchmark process. When the benchmarks are completed, the results are available in the Job Storage Triplestore.

7.2 Future Work

The development performed in this thesis has resulted in a running version of the Basilisk platform. Currently, the benchmark process for Docker Hub repositories is working, and benchmarks are successfully performed.

Because the thesis time schedule had to be altered during the implementation process, some functionalities have not been fully implemented. Most of these missing functionalities are not strictly relevant to successfully running the platform but would be desirable from a user perspective.

These functionalities are, for example, a user management system, extended triplestore configurations, and the Basilisk frontend. The user management system could manage different user roles and access rights for the various system functionalities. For example, some users could be only allowed to view the job status or create a manual job, while admin users could change the triplestore configurations. The extended triplestore configurations could define a range or list of versions for which a configuration is valid for use. If, for example, a newer triplestore version requires the dataset to be loaded differently, a new configuration could be set up to be used with that version. Lastly, the Basilisk frontend would provide a fast and easy way to interact with the benchmark results and could also offer a user interface for setting up and managing the configured triplestore and benchmarks.

The most important functionality that is still missing in the platform is the execution of benchmark jobs on GitHub repositories. As stated in section 5.1 the functionalities for observing GitHub repositories and creating benchmark jobs are already implemented. What is still missing is the functionality for downloading the source code from GitHub and building a Docker image from the source code files. After that, the process of starting the image and running the benchmark will be the same as with Docker Hub repositories.

Bibliography

[1] Iguana Docs - Metrics, Retrieved Nov 23, 2021, http://iguana-benchmark.eu/docs/3.3/usage/metrics/.

[2] RDF 1.1 Concepts and Abstract Syntax, Retrieved Jan 28, 2022, https://www.w3.org/TR/rdf11-concepts/.

[3] RDF 1.1 Turtle, Retrieved Jan 28, 2022, https://www.w3.org/TR/2014/REC-turtle-20140225/.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] Noah Gibbs. Microbenchmarks vs Macrobenchmarks; Retrieved May 6, 2022, https://engineering.appfolio.com/appfolio-engineering/2019/1/7/microbenchmarks-vs-macrobenchmarks-ie-whats-a-microbenchmark.

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

[11] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language, Retrieved Dec 12, 2021, https://www.w3.org/TR/sparql11-query/.

[12] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph, and York Sure-Vetter, editors. *Semantic Web: Grundlagen*. eXamen.press. Springer, 1. aufl edition.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

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

List of Acronyms

HCS	Hooks Checking Service
JMS	Job Managing Service
TBS	Triplestore Benchmarking Service
JRQ	Job Request Queue
BJQ	Benchmark Job Queue
JSTS	Job Storage Triplestore
SWDF	Semantic Web Dog Food
LUBM	Lehigh University Benchmark
SP²Bench	SPARQL Performance Benchmark
WatDiv	Waterloo SPARQL Diversity Test Suite
RDF	Resource Description Framework
IRI	Internationalized Resource Identifier