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, February 2, 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

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[16].

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 [16]. IGUANA is a benchmark-independent execution framework [8] 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. Many of these existing benchmarks focus on different goals and scenarios to test the Triplestores. 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[10]. These synthetic benchmarks have the advantage, that they can be generated to arbitrary sizes. The main point of criticism is that the generated scenarios can easily be abstract and not representative of a real world situation [15].

The LUBM Benchmark[10] is a synthetic benchmark which focuses on the reasoning and inferencing capabilities of the Triplestores under test. The test data is about 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[17]. 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[15]. They also test for SPARQL features like union and optional graph patterns.

The WatDiv suite generates a synthetic benchmarks and consists of multiple tools[6]. 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 Union and Optional patterns.

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[13, 15].

FEASIBLE is a benchmark generation framework which generates datasets and queries from provided query logs[15]. 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, since those environments are specialized for the given benchmark and are not easily interchangeable[8].

The next sections focus on benchmark-independent execution frameworks.

2.3.1 IGUANA

IGUANA is a SPARQL benchmark-independent execution framework[8]. 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 IGUANA.

review

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[14]. 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[14]. The data for benchmarks has to be stored in docker containers, 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, which is not really efficient.

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. Basilisk focuses on a smaller use-case of benchmarking SPARQL endpoints continuously.

review

[1] `https://azure.microsoft.com/`

[2] `https://aws.amazon.com/`

3 Background

This chapter explains the fundamental topics required to understand 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 of the context in which the Basilisk platform is 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[12].

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.

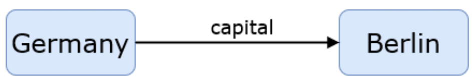

Figure 3.1: Simple Knowledge Graph

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 is a framework for describing data and knowledge in a standardized way [4] and 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 [5] 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[7], GraphDB[1], Virtuoso[2], or Jena TDB[3].

This thesis focuses on Triplestores that 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

review

SPARQL (SPARQL Protocol and RDF Query Language)[11] 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.

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.

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 goal of the Basilisk platform. The sections Microservice and

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

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?capital
WHERE {
        dbr:Germany dbo:capital ?capital .
}

---

SELECT ?relation
WHERE {
        dbr:Germany ?relation dbr:Berlin .
}
```

Figure 3.3: SPARQL query examples

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 data set and a set of operations or queries which will be performed on the data set. 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[1].

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, which will be set up in this thesis, will only perform 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 [9]. All Microservices can be individually deployed and managed and 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 [9, 2]. 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[3].

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.

review

3.2.6 Software Design Patterns

Repository Pattern

fill

Domain Driven Design

fill

[5] `https://spring.io/`

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

4 Approach

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

The main idea for the Basilisk platform is that a user can register a Triplestore for a continuous benchmark by setting up a hook to the repository on GitHub or a docker image from Docker Hub containing the Triplestore, which will then be observed by Basilisk. If there is a new release of the Triplestore, Basilisk will fetch and build the new docker container to perform a benchmark. The measured results of the benchmark will be stored in a database and are then available through the web frontend to review.

right position for this explanation?

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 a message queue system.

Figure 4.1 gives an overview of the microservice architecture for the Basilisk platform and the most important messages send between the services.

nic

TODO: better graphic

Figure 4.1: High level design of the Basilisk framework

4.1 Main Services

The next sections explain the three main services, namely Hooks Checking Service (4.1.1), Jobs Managing Service (4.1.2), and Triplestore Benchmarking Service (4.1.3).

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

4.1.1 Hooks Checking Service

The main task of the Hooks Checking Service (HCS) is to observe Github and Docker Hub repositories 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) needs to be observed for changes. This happens through API calls from the frontend to the hooks checking service 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 to regularly check for new versions and releases. Each hook stores the latest known version of an repository to easily compare for new versions.

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

API

4.1.2 Jobs Managing Service

The Jobs Managing Service (JMS) processes the requests coming from the web-frontend, checks if the Hooks Checking Service has found a new version for a benchmark and creates jobs for new benchmarks.

API

4.1.3 Triplestore Benchmarking Service

Lastly the Triplestore Benchmarking Service (TBS) executes the benchmarks given to it and saves the results to a database.

API

4.2 Message Queue System

The messages send between the services are transmitted via the RabbitMQ[1] message broker software.

4.3 Programming Language and Frameworks

All services are implemented with Java and are using the Spring Boot framework.

4.3.1 Package Structure

The package structure used for the service implementation is similar in all three services. It is strongly influence through the use of the Spring Boot framework. The Spring and Spring Boot framework are shortly described in chapter 3.2.5. In the following the most important packages of the services are described.

[1] `https://www.rabbitmq.com/`

config In the config folder important Spring Beans are defined and configured. Also the configuration for the RabbitMQ message queues is defined here. The classes in this package are marked with the @Configuration annotation which informs the Spring framework that these classes declare @Bean methods for generating Spring Beans. Beans are defined in Spring as the objects that form the backbone of the application.

core

4.4 Frontend

5 Implementation

5.1 Implementation TODO

- Extend APIs
  - delete hooks
- Implement benchmarking using IGUANA

6 Evaluation

- Experiment setup, requirements - Performing of benchmarks - Result evaluation

7

Summary and Discussion

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

Bibliography

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

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

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

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

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

[6] 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.

[7] 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.

[8] 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.

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

[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, https://www.w3.org/TR/sparql11-query/.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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