A Framework for Reproducible Evaluation of Semantic Storage Options

Jedrzej Rybicki* and Benedikt von St. Vieth§
Juelich Supercomputing Center (JSC)
Forschungszentrum Juelich GmbH, Germany
Email: *j.rybicki@fz-juelich.de, §b.von.st.vieth@fz-juelich.de

***Abstract*—Despite the urgent need to conduct computer-driven experiments in a reproducible fashion, there is no widely-accepted, established approach to this problem. The rise of Open Data surely helps in understanding and verifying some of the research findings, but the true verification comes from a means to reproduce the original tests. This paper proposes a framework for conducting such reproducible experiments. It leverages Docker to facilitate exchange of, not just source code used in test trails but rather whole, ready-to-run experimental environments. The lightweight virtualization provided by Docker makes it very portable across a wide variety of hardware and software platforms. We explain the framework in detail, discuss its advantages as well as shortcomings, and assess how future-proof it is. The usability of the proposed framework was verified by conducting a reproducible evaluation of the possible storage options for semantic annotations. This use case emerged in the context of a particular distributed infrastructure, where users requested a possibility to annotate stored digital objects. There are many possible technologies that can be used to store semantic annotations. We evaluated performance and suitability of relational databases, document stores, and graph databases, to conclude that the graph database is the best candidate to efficiently handle the task. Although, it has also some limitations, which we will point out. By using the proposed framework in the aforementioned evaluation, we enable other researchers to repeat it, but also benchmark alternative technologies if required. Furthermore, the framework can be reused in an iterative process of performance tuning of the selected storage option, to quantify and verify the influence of particular configuration changes. The main output of this paper is a framework which allows to easily redo the evaluation of semantic storage options.**

Keywords–Deploying Linked Data; Reproducibility; Distributed Infrastructures; Benchmarking.

I. INTRODUCTION

This work is an extended version of our previous publication [1], it presents new experiments, new results, and a more detailed literature review.

Distributed research infrastructures like EUDAT (EUropean DATa [2]) provide generic services to manage research data in an efficient and cost-effective way. Since the research communities using the services advance over time, they are constantly expressing new requirements with respect to kinds of data and possible usages that the infrastructure should support. An example of a community requirement EUDAT, was confronted with, was the support for semantic annotations across its data management services.

Semantic annotations are a very powerful tool to work with data in a distributed environment. They extend the context of the data, and by that increase the data understandability. One can conjure the semantic annotations as a facility to add metadata and comments to entities managed in the infrastructure. An example would be a keyword attached to a digital object, but more sophisticated cases are envisioned as well. We will explain the model in more detail later in this paper, but astute reader can imagine that efficient annotations handling should enable different types of search queries. It should be possible to retrieve all annotations for a given object, but also reverse lookups (i.e., localizing all data objects with given keyword in our example) will be used. The uptake of this new service will only happen if sufficient performance of both kind of queries can be granted.

This paper is focused on evaluating semantic storage options. We consider a technology to be a valid option for storing annotations if it allows for permanent storage and very basic retrieval operations as described above. There are a lot of products on the marked that advertise much more sophistication with respect to handling specific semantic operations but we wanted to remain generic and include broad range of products in our evaluation. There are many ways in which annotations can be stored. The EUDAT service plans to use the World Wide Web Consortium (W3C) Annotation Data Model [3]. As it is based on JavaScript Object Notation for Linked Data (JSON-LD [4]), an obvious approach would be to use document stores for the task. Such storages (also called document-oriented databases) are optimized to handle semi-structured data (called documents). The managed documents are usually in JSON format. Examples of document stores are MongoDB [5], CouchDB [6], or Elasticsearch [7].

Because annotations are attached to the data objects, the whole data set forms a graph with managed entities and annotating metadata as nodes and annotations as relations between them. Thus, we have included the graph database neo4j [8] into the evaluation of possible storage backends. Lastly, based on the feedback to the original conference paper describing our first results [1], we include relational database management system (RDBMS) MySQL [9] into our evaluation. Relational databases are mature technology, their operations are well understood, and they have high proliferation in distributed infrastructures. Therefore, they constitute an excellent reference point for the above mentioned (perhaps less known) alternatives.

Requirements submitted to EUDAT are analyzed from different view angles. One of them is the performance evaluation of candidate technologies. To this end, resource and service providers are constantly testing and benchmarking possible approaches and new technologies. Such evaluations, must adhere to scientific standards in terms of methodology, transparency, and reproducibility. This is absolutely necessary to make the transparent decisions whether or not a given requirement is implemented and how. Furthermore, such evaluation can be reused by other infrastructures and service providers. The trend to share the results subsumed under the term of Open Data [10]

is an important first step in sharing the results. But much more beneficial would be to move the sharing beyond just the results and to make the programs and experimental setups sharable as well. Currently, there is no established solution for such a sharing. The paper includes the results we obtained and their discussion. But our overarching goal was to make the result reproducible, i.e., provide all the tools we used in our evaluations in such a form that an independent researcher can not only retrieve and analyze them but also redo the evaluation and obtain the same results (at least quantitatively). We limit ourselves to the software layers of the conducted experiments starting from the used operating system, libraries, up to the software used for generating the data, measuring the scalability and visualizing the results. This working definition of reproducibility we use across the paper lays somewhat between terms reproducibility and replicability as used in science philosophy.

In one of our previous papers [11], we have already shown that Docker [12] can be leveraged to provide on-demand instances of popular web services in the context of a distributed research infrastructure. Although, such seamless provisioning of services can be used to conduct reproducible research, there are more aspects to it. In this paper, we will exercise a whole workflow from testing, through result processing, up to visualization of the outcomes. We will use Docker and `docker-compose` [13] to conduct the steps in a transparent, sharable, and reproducible way. We will test our approach by evaluating storage options to handle semantic annotations. Based on our results, one can select a technology to best fit her particular use case. The selection of the technology is, however, only the first step towards a working solution. The software used to manage the data must be tuned to obtain best possible performance. Such a tuning is usually done in an iterative way, where the influence of each particular change in configuration on the overall performance is measured and evaluated. The reproducibility framework presented in this paper makes such iterative tests easier to run.

This paper is an extended version of a conference contribution [1]. It includes more experiments and a new possible technology candidate (MySQL). Beside providing more details on the our testing framework, we devoted an additional section to a discussion of functional suitability of the selected technologies. In particular, we elaborate on how the JSON-LD model can be stored in different backends. A new feature of our framework, which we describe in detail here, is the possibility to orchestrate all the experiments and run all the tests, process the results, and visualize the results in a fully automated fashion. This was a remaining step for seamless sharing of the complete experimental setups. We also include an assessment of how future-proof our solution is and a detailed review of the related work.

The rest of this paper is structured as follows. In Section II we will review the state of the art of both semantic storage options and reproducible evaluations in computer science and lay down our vision. Section III explains what semantic annotations are and discuss suitable storage options. Subsequently, we present the selected storage technologies in more detail and touch on the related subject of data modeling. Section V is devoted to the detailed experimental setup and we describe how our reproducibility framework is built and should be used. In particular, the section also discusses how future-proof the approach is. Section VI is presenting the results for the selected semantic storage options. Detailed discussion of the evaluation of the different storage options can be found in Section VII. We conclude the paper with a summary and an outlook on the future work.

II. Related Work

De Witte et al. [14] prepared a benchmark for evaluating triple stores to store Linked Data. Interestingly, they rely on official images available in the Amazon Web Services Cloud [15] to make the results of the evaluation more reproducible. In the later work of this authors also approaches of running the benchmarks with help of Jupyter Notebooks [16] resemble some similarities with our framework. In our work we concentrated more on semantic annotations (with different benchmarks) and did not include triple stores in our evaluations. Also our focus was on making the complete evaluation reproducible beyond particular cloud solutions.

Pacaci et al. [17] examined applicability of relational databases to store social-media-like graphs and compared their performance with graph database systems. Their results are similar to ours. The advantage of the graph database is very good visible for shortest-path queries. In terms of point queries, relational database systems perform better. Also in terms of the writing performance the results are pretty analogous. The relational database (PostgreSQL [18]) exhibited significantly better update performance and maintain up to an order of magnitude faster write throughput compared to their competitors. It should be mentioned that the experimental setup used there differs significantly. In the referred work, unlike our experiments, the whole database are loaded into memory. We believe that this is very beneficial for the relational databases as joins become much faster. The results surely speak for including relational databases in our evaluation.

Hernandez et al. [19] used Wikidata [20] knowledgebase to define a set of benchmarks for popular semantic storage solutions. The focus of the work lays on creating a benchmark with queries close to the observed in real life deployment. It would be interesting to combine the approach into the presented experimental framework to produce more meaningful results. With respect to benchmarks it is worth to mention one more effort. The Linked Data Benchmark Council (LDBC) [21] is a joint effort of academia and industry devoted to establishing benchmarks, benchmark practices, and benchmark results for graph data management software. It has developed two benchmarks: Social Network Benchmark [22] and Semantic Publishing Benchmark [23]. Even if the proposed benchmarks might not necessarily be the best indicators whether a selected technology can provide good performance when storing semantic annotations, they give a lot of useful hints in this regard. It would be also very interesting to incorporate this standardized benchmarks into our experimental framework, we might pursue this direction in the future. The final result should be a standardized set of benchmarks and a agreed-on framework for conducting reproducible experiments using the benchmarks.

There exists a body of work on reproducible research in computational science. Donoho et al. [24] provide an interesting philosophical background on the problem of reproducible research, underlining the potential credibility crisis lack of it can lead to. Finally, they describe their approach to

reproducibility research in harmonic analysis. The approach are developed for different kinds of problems, and use different (discipline-specific) tools. Freire et al. [25] provides an overview of the state-of-the-art of reproducible research in the context of data management. Another example of a community defining standard practices for sharing computer code and programs is the publication of Eglen et al. [26], discussing the issue in the context of neuroscience. It might be expected that more and more research communities will undertake the effort of explicitly defining best practices and tools for reproducible experiments. Our work is parallel to such efforts. The proposed framework is not tailored to a specific community. It can, however, accommodate many community-specific tools and thus serve as a basis for the community specific standard testing procedures.

The problem of reproducibility is closely related to the idea of research collaboration. Chandy et al. [27] present how experiments can be conducted in a fully distributed fashion. Where both resources and researchers are spread across the world, yet still are able to conduct reproducible research. We believe that our work, conducted in context of a distributed, inter-disciplinary infrastructure offered by EUDAT, fits well in the authors' vision.

III. Semantic Annotations

Roughly speaking, the EUDAT distributed infrastructure is built to manage research data in form of digital objects (DOs). The objects are composed of bit streams and uniquely identified by persistent identifiers (PIDs) [28]. In some cases the objects also include metadata descriptions. The PIDs are offered by the ePIC system [29]. They can be used to reference the particular objects, for instance in scientific publications. The PID system constitutes an indirection layer that simplifies (future) access to the digital objects, also allowing to move the data between locations without invalidating publications referring to them.

On the other hand, PIDs are just opaque identifiers and on their own provide very little information about the DO they are pointing to, hence, they are not very well suited for browsing and searching through the EUDAT data repositories. Therefore, EUDAT is working on enabling semantic annotations for the objects stored in its distributed research infrastructure. Such annotations describe the DO they are pointing to, extend their context and thus have the potential to facilitate efficient search of data. The current approach is to use the W3C format for annotations. The W3C web annotation data format is pretty simple: Each annotation is a relation between a *body*, e.g., EUDAT data object, and *target*, e.g., metadata describing that object. A basic annotation, as proposed by W3C, is shown in Figure 1.

It is important to notice that both target and body objects in the annotation have unique identifiers. These are crucial from the user's perspective. It can be expect that the users will be interested to view a list of all annotations for a given body id, i.e., all metadata descriptions for a specific data object. Also, a reverse lookup producing all the data objects with specific tag (i.e., a retrieval by target id) represents important functionality. Those expected usage scenarios were used as major hints for our benchmarks. We use three kinds of operations to mesure suitability of a semantic storage option:

- creation of a new, non-existing annotation,

```
{
  "@context": "http://www.w3.org/ns/anno.jsonld",
  "id": "http://example.org/anno2",
  "type": "Annotation",
  "body": {
    "id": "http://example.org/analysis1.mp3",
    "format": "audio/mpeg",
    "language": "fr"
  },
  "target": {
    "id": "http://example.gov/patent1.pdf",
    "format": "application/pdf",
    "language": ["en", "ar"],
    "textDirection": "ltr",
    "processingLanguage": "en"
  }
}
```

Figure 1. A Basic Annotation in W3C Annotation Data Model Format (source: [3]).

- retrieval of an existing annotation by its target id,
- retrieval of an existing annotation by its body id.

All those operations are expected for the final service offering the storage and management of semantic annotations in EUDAT infrastructure. The ratio between the operations is not yet known, but it can be expected that retrieval will be more frequent than store operations. In the long term, more annotations will be read than written.

IV. Storage options

There are many options to store semantic annotations. One obvious approach would be to stick to the JSON-LD rendering as proposed by W3C, use it as the internal storing format, and find a storage backend that can support it. There are many NoSQL solutions (called document stores), which are optimized to manage JSON documents. MongoDB [5] is one of the most popular document stores on the market.

Another storage option we considered is based on the following observation. Annotated objects with annotating metadata form a graph (with annotations as edges). To account for this way of thinking, a graph database like neo4j [8] could be used as a storage backend.

Finally, we included a relational database representative MySQL [9] in our evaluation. It is a mature technology, familiar to both system administrators and users. Although its direct suitability will be discussed later, we believe that a traditional relational database can server very well, at least as a reference point for the remaining results.

For our experiments we used a very simplified form of annotations. We reduced them to just having a target, body id, and creation time. So they were in fact much simpler than the one presented on Figure 1, yet sufficiently complex to approximate the real-world usage. Thus, there were no problems in storing the annotations in the selected database backends. The question of the performance of the products will be discussed later. Firstly, we would like to discuss the suitability of the single products in terms of their functionality, i.e., whether or not they can handle more complex annotations.

The JSON-LD "internal representation of an annotation is the result of transforming a JSON syntactic structure into the

core data structures suitable for direct processing: arrays, dictionaries, strings, numbers, booleans, and null" [4]. Modeling arrays and dictionaries in a relational database might be hard. Two obvious options are to store the content of an array as concatenated strings or store the content in a separate table and subsequently use joins to render the complete representation of an annotation. Former approach has its obvious limitations. String comparisons are costly, and for more sophisticated searching criterion, even more expensive regular expressions will have to be used. The later approach, comes at the cost of performance penalty of joins. Similarly, storing "internal" JSON objects, encapsulated in JSON-LD is not an easy task for a relational database. It would probably require to again use distinct tables for each of such an internal objects and rely on laborious joins to render the complete annotation object.

The relational databases require rigid database schema defining fields of single tables, whereas JSON-LD enables far-reaching flexibility in this regard. Some annotations might have fields that other annotations do not have. Depending on the application it might be possible to pre-define a set of annotation fields or again use joins to make the schema more flexible. As mentioned above, each join influence the performance in a negative way. It also makes the queries required to gather all the data from all tables more complex. For our experiments we have modeled annotations as a single table in the relational database. With a primary key composed of target and body id. Because we avoid joins, we obtain lower limit of the response times. Each join, required by more sophisticated models, would increase the response times.

MongoDB is a document store, which supports JSON natively and, therefore, it can handle JSON-LD without additional adaptation. It does not require (and in fact does not allow) to define an equivalent of a database schema. The JSON documents (i.e., annotations) stored in one database can differ significantly with respect to fields they contain. This high flexibility comes at a cost. Although, an explicit, up-front schema model is not required, an implicit model (at the query time) is still present. The application logic has to get to grips with potential heterogeneity of the records, for instance, when presenting the results to the users. Also the lack of a schema makes the definition of constrains and, thus reduction of redundancy, harder. Each document in the document store contains a complete annotation with both target and body objects. The most popular keywords will be stored many times. Such redundancy is relevant for the typical kind of queries the database would have to handle. For instance, to identify all the data objects marked with a given keyword, a full scan of the database would be required. Also, to avoid duplicates, a search would have to be conducted before a new item is inserted into the database. In our experiments, we don't create redundant annotations, the benchmarking program takes care of generating unique content, and we see no redundancy problems, which would occur in real-world applications.

The graph database we used in our experiments (neo4j) does not support JSON-LD natively, but it does not require a rigid schema either. It is possible to have graph nodes with a varying list of fields. At the same time, neo4j occupies a middle ground position with respect to defining constrains. It is possible, yet not mandatory, to define fields that each annotation has to have. It is also possible to define uniqueness constraints to ensure that no duplicates of annotations are stored. We have modeled annotations as two nodes connected by a single relation. In the labeled graph model that neo4j implements, it is possible to add properties to nodes and relations. Properties can be defined as lists and maps so there is no mismatch between JSON-LD and the neo4j model.

V. Experimental setup

To obtain meaningful benchmarking results it is important to minimize the number of "moving parts" and reduce the testing environment to components, which are absolutely necessary. In particular, we were not interested in the performance of the web interface that will be used to work with annotations or the performance penalty caused by its integration with other EUDAT services. Therefore, we have written a Python program with methods for generating annotations with unique body and target identifiers, and for storing and retrieving of the data. The methods use simple interfaces to access selected database stores: MongoDB, neo4j, and MySQL.

A. Docker

To enable easy reproducibility of the conducted tests, we have prepared a Docker-based environment. Docker [12] is a lightweight virtualization solution based on Linux Kernel features like *namespaces* and *cgroups* to isolate guests from the host system. Docker uses image templates to start containers (i.e., guest processes). Furthermore, Docker provides tools to easily exchange images via the public Docker Hub [30], or private on-site repositories. Docker introduces a notion of an official image, which is created and maintained by the provider of a given technology. There are official images for major Linux distributions, but also for popular content management systems and databases. It is possible (and common) to take such images as basis, modify them (e.g., by installing software, or changing their configuration), and publish them as new images in the Docker Hub. The images are built in a hierarchical fashion by applying a "write-on-modify" principle. Thus, it is possible to trace back all the changes done to a given image during the installation and configuration of the software it comprises. It is also possible to review the content of an image before running it. This significantly increases the mutual trust between Docker users and reduces security implications of running Docker containers.

One way of creating images, which we used in our work, is to create a `Dockerfile` describing all the steps required to setup the dependencies, install the software, and configure it. An example of such a `Dockerfile` is depicted in Figure 3. It is an image file of our testing program. Readers with some Debian/Ubuntu experience will recognize initial steps of installing, e.g., `python`, starting from the 3rd line of the file. A sequence of following `RUN` commands set up the python dependencies required by our testing program. The `CMD` directive is defining a command to run upon creation of a container. By default, it will run a bunch of tests against all the supported database backends. The command can be overwritten for each container created from this image. The `VOLUME` instruction states that `/results/` directory in the containers created from this image will store unique data (in our case results) and is not part of the image. As we will show later, upon creation of a container, it is possible to map the volume on any directory of the host machine.

```
mongo:
  image: mongo:3.4.5
neo:
  image: neo4j:3.2
  environment:
     - NEO4J_AUTH=none
mysql:
   image: mysql:5.7.18
   environment:
      - MYSQL_ROOT_PASSWORD=root
tester:
  build: .
  dockerfile: Dockerfile-tester
  links:
    - mongo
    - neo
    - mysql
```

Figure 2. An example of the `docker-compose` deployment descriptor for our experimental environment.

The Docker ecosystem embraces many tools, among them `docker-compose` [13], which is a tool for defining and running multi-container applications that we are going to use in our experiments. In our case we run the testing program and respective backends. The deployments for `docker-compose` are defined via `yaml` files, describing what containers should be started and how they are connected to each other. The deployment descriptor for our testing environment is depicted in Figure 2. It defines three containers for the storage backends, and how they should be created (by using official images). Finally, a container with our benchmarking program is defined. It is created from a `Dockerfile` (`Dockerfile-tester`) and is connected to the remaining containers. For Docker images it is possible to define explicit versions that shall be used. Usually also an image with a tag `latest` is available, and points to the most up-to-date version of the given product. As can be seen in Figure 2, we use explicit versions of the images.

There are many reason why we are using Docker as a basis of our framework. Firstly, due to the virtualization it is possible to run our test programs on almost any platform (regardless of the operating system it uses). The images also contain the dependencies and libraries required, so again the configuration of the host system may be neglected. The possibility to review all the changes done in a particular Docker image enables transparency and understandability of the obtained results. The Docker Hub facilitates an easy exchange of the Docker images containing programs used in the evaluations between researchers. Last but not least, by using Docker volumes, it is possible to separate data from the programs, in our case: results and processing tools.

B. Solution details

All technology providers we considered (MongoDB, neo4j, and MySQL) offer official images for their databases, which we used for our evaluation [31] [32] [33]. We created a Docker image for our testing program and prepared a `docker-compose`-based testing environment. The source code and the documentation is stored on GitHub [34], enabling the verification and repetition of the benchmarking. In fact, we plan to reuse this framework to do some further testing of different EUDAT-inspired use cases in the future. Given a system with a running Docker daemon and `docker-compose`, it is first required to build, and retrieve official images.

```
docker-compose build .
docker-compose pull
cd processor
docker build . -t processor && cd ..
cd visualizer
docker build . -t visualizer && cd ..
```

Starting the evaluation benchmark is done by merely issuing one command like:

```
docker-compose run \\
  --volume=/path/:/results/ \\
  --name exp1
  tester
```

The `--name` parameter of the above command is not strictly required, it attaches a user-defined name (`exp1`) to the particular experimental run, which is convenient for the further analysis. The `--volume` parameter maps the `/results/` directory from the container on `/path/` directory on the host system. Hence, it is possible to separate results for different experiments from each other. It only requires to map the container volume to different physical paths of the host.

Also, Docker images for processing of the results and visualizing them are provided. The first step transforms the results from the evaluation by using command:

```
docker run --volumes-from exp1 \\
  processor
```

The processing step is transforming raw results from the first step into a more human readable form. If multiple results for a given set of parameter are available, an average value with standard deviation is calculated.

The `--volumes-from` parameter is used to attach the storage volume with the data produced in the first step to the newly created Docker container. Please note that we are using the name assigned to the experiment in the previous step (`exp1`) rather than a physical path on the host system. It should be stressed that for a volume of a container to be used by different container, it is not required for the former to be running. In our case, at the time of processing of the results, the benchmarking program is no longer running. Docker maintains its volumes, unless they are explicitly deleted by the user. So there is some persistence of the volumes beyond the lifetime of single container (and experiment run).

Finally, the plots that we will present in the following section are created with help of `gnuplot` [35] and other tools embodied in a Docker image, which again uses volume with data from previous steps and can be run with following command:

```
docker run --volumes-from exp1 \\
  visualizer
```

To enable the sequential processing of the data, we internally agreed to store all the data (results, visualizations, etc.) in the same path defined as a Docker volume. Thanks to this contract, we can guarantee that data are not becoming part of the Docker images and thus will not hinder their

```
FROM ubuntu:latest
MAINTAINER j.rybicki@fz-juelich.de
RUN DBEIAN_FRONTEND=noninteractive apt-get update && \
   apt-get install python python-pip python-mysql.connector -y && \
   apt-get clean autoclean && \
   apt-get autoremove && \
   rm -rf /var/lib/{apt,dpkg,cache,log}
VOLUME /results/
RUN mkdir /app/
ADD . /app/
RUN pip install -r /app/requirements.txt && chmod +x /app/test.py
WORKDIR /results/
CMD sleep 10 && \
    /app/test.py dummy 10 1000 && \
    /app/test.py neo 10 1000 &&  \
    /app/test.py mongo 10 1000 && \
    /app/test.py sql 10 1000
```

Figure 3. `Dockerfile` of the Tester image.

reuse. Secondly, it is easily possible to extend the workflow by adding new steps or modify existing ones, for instance, if different types of visualization are required. It must be, however, safeguarded that the processing steps don't overlap their outputs, e.g., don't use the same file names to store the results.

C. Orchestration

Although it is hard to quantify, we found Docker tools pretty easy and intuitive to use. In fact, not much deep knowledge of Docker is required to just run the above commands. Nevertheless, our goal is to make the results highly and easily reproducible. We aim at ways of sharing complete, running, testing environment with all the dependencies required to repeat the tests.

Docker does not support the execution of workflows (i.e., subsequent commands). Thus, we had to search for alternatives. One solution would be to use scripts or programming languages to automate the execution of the commands. It is not an easy task to make the scripts understandable, adjustable, and really portable. It would require a lot of effort to create scripts for all the platforms on which prepared Docker can be deployed. Alternatively, the scripts could be encapsulated in a virtualization solution to enable cross-platform compatibility. In fact we already presented a lightweight virtualization solution, namely Docker.

It is possible to chain commands as mentioned before in a `CMD` directive of a new Docker container, which we call orchestrating container. It would not be, however, very beneficial to then create the testing environment in the container as this would substantially increase the virtualization overhead. Therefore, we propose a different solution. The orchestrating container could access the Docker interface of the host it is running on, use this interface to create the testing environment, run the tests, process the results, and produce the visualizations. An overview of this approach is depicted in Figure 4. The socket file used for communication between Docker client and Docker daemon can be injected into the orchestrating container and subsequently used by the Docker and `docker-compose` clients installed in the orchestrating container. After starting the container it will pull all required images from Docker Hub and afterwards run them in the proper order. Therefore, to execute the whole workflow, it is only required to issue just one command:

```
docker run \\
  -v /var/run/docker.sock:\\
  /var/run/docker.sock \\
  httpprincess/orchestrator:1.0
```

The main disadvantage of the solution is its lack of flexibility in terms of managing volumes. Volumes used for storing the results are mounted from the host machine and not from the orchestrating container. Thus, to change the location the `/results/` volume is mapped on, one would have to manipulate `docker-compose` setup files confined in the orchestrator image or use the default location. It is still possible to refer to the volumes by using container names though. Also, to change the evaluation parameters it would be required to change the image. To this end, a feature of running Docker containers in an interactive way might be beneficial.

We expect that the provided orchestration container will be used to automate the parameter tuning or for just repeating the experiments we did. For more advance usages (like testing against different technologies), the orchestrator will provide hints on how to conduct the experiments and setup the testing environment, but the researcher will have to step in and make some changes anyhow.

D. Making the framework future-proof

Our main goal was to make the evaluation of the semantic storage options reproducible. It also means that the researchers wishing to repeat our experiments should be able to do so even in some distant future. In this section we would like to discuss how future-proof the presented solution is.

It is hard to predict changes in the technology landscape. The main corner stone of our framework is Docker. Although this technology is pretty new, it is based on Linux Kernel features available already for a longer period of time. It should be stressed that Docker (unlike typical virtualization technologies) does not include a Kernel in the containers but rather use the Kernel of the host system and rely on its features for process isolation. The Docker images we use as a basis for our images will become obsolete, but by publishing our `Dockerfiles` we enable the rebuilt of the

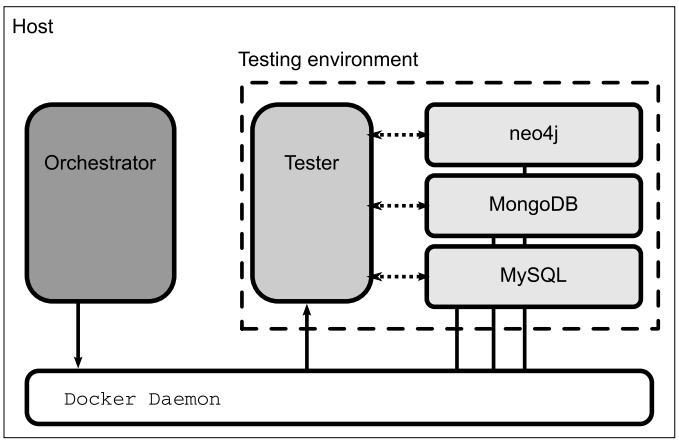

Figure 4. Schematic explanation of the orchestration solution.

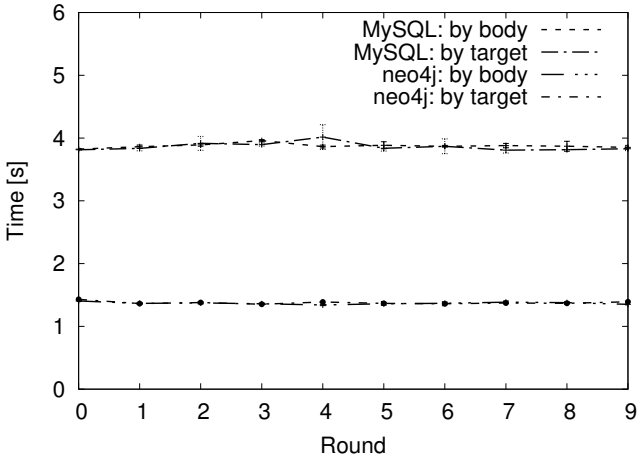

Figure 5. Difference between retrieval by target and by body for neo4j and MySQL ($reps = 10000$ records are added and retrieved in each round).

images with new versions. Using just the `Dockerfiles` to built the images, has some drawbacks. As can be seen in Figure 3, in the process of building the image, software from the official Linux distribution repository is retrieved and installed. When (in not so distant future) new versions of this software become available, the produced images will also be different. To circumvent this limitation we uploaded the images to the Docker Hub [36] [37] [38] [39]. It is no longer required to build the images, they can be retrieved from this repository with a `pull` command:

```
docker pull httpprincess/tester:1.0
```

Docker allows for tagging of the images with versions. Even if the images change in the future, and new version becomes available, it is still possible to retrieve old images. All the images used for the experiments presented in this paper were tagged as version `1.0`.

When discussing a more distant future we run into different kinds of issues. If Docker become obsolete, the availability of our source code [34], together with the respective `Dockerfiles` becomes important. Although the images cannot be built anymore (presuming there is no Docker), they still constitute a formal description of the installation and configuration process.

The future availability of the selected storage technologies is not in our hands, but this is the motivation for making the process of performance evaluation reproducible. The same tests can be run against new version of the products or (after some modifications) against similar products that will become available. As mentioned before, in the `docker-compose` file describing our testing environment, we use explicit versions of the official images of the storage technologies (rather than opting for `latest` images). As long as those versions are available in the Docker Hub, it would be possible to repeated our experiments.

VI. Results

All the tests were run on the same virtual machine with 16 VCPUs, 16 GB RAM, using Ubuntu 16.04 LTS. We used official Docker images for MongoDB in version 3.4.5, neo4j in version 3.2, and MySQL in version 5.7.8. The Docker daemon was running version 1.12.6, and `docker-compose` in version 1.14.0.

Our experiments are defined by three parameters:

1) $engine$: database engine (currently MongoDB, neo4j, and MySQL),
2) $rounds$: number of rounds,
3) $reps$: number of repetitions in each round.

The tests were divided into rounds and in each round all the previously defined database operations (see Section III) were conducted in the following order. Firstly, $reps$ number of records were created, subsequently random (with repetition) $reps$ annotations were retrieved by specifying existing $target.id$, finally $reps$ random annotations were fetched by $body.id$. We measured the time of each activity, that is the complete time to create records, time to retrieve all $reps$ record by target and body id. Three time measurements were made in each round. Please note, that no records were removed, i.e., for given $reps = 1000$, the database grown in each round by new 1000 records. If not stated differently, each experiment was repeated three times and the average value of response times with standard deviation were calculated for each round.

A. Retrieval scalability

Figure 5 presents the response times for retrieval of the annotations by target and by body. We only compare the results for neo4j and MySQL. As can be seen, for a given technology there is not much difference in the response times. Situation looks a little bit different for MongoDB, which is depicted separately in Figure 6 as the absolute values are much higher, and would make the previous plot less readable. There is a difference between body and target retrieval, yet the differences lay within the standard error value. For the reminder of this section (if not stated differently) only retrieval by the body id will be presented.

In Figure 7, Figure 8, and Figure 9, we depicted the retrieval scalability of each technology tested. For that we conducted three experiments with different values of $reps$ (1000, 5000, 10000), each had 10 rounds. Figure 7 shows

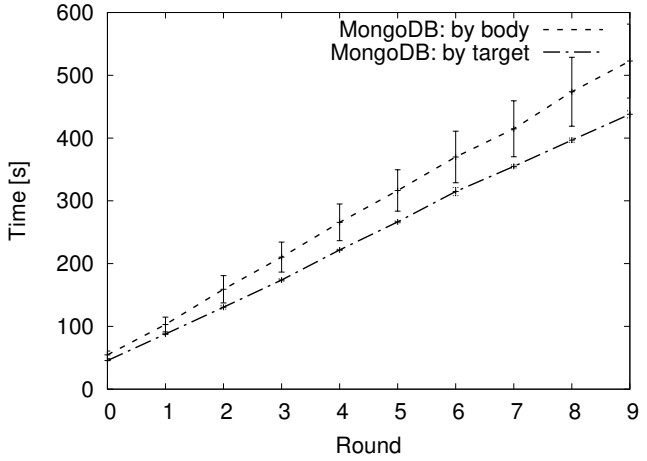

Figure 6. Difference between retrieval by target and by body for MongoDB ($reps = 10000$ records are added and retrieved in each round).

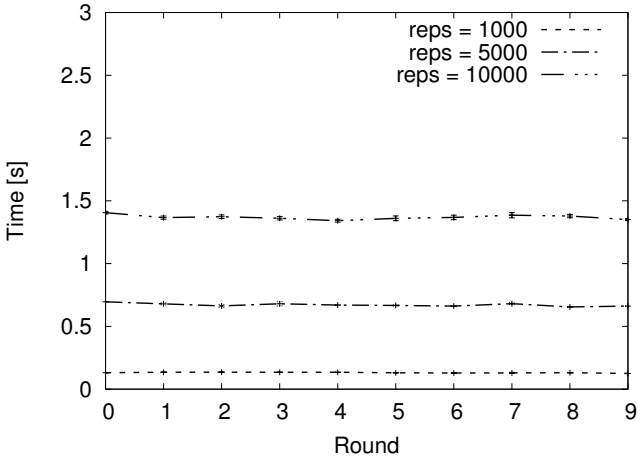

Figure 8. Retrieval scalability for neo4j ($reps$ new records are added, and $reps$ random records are retrieved in each round).

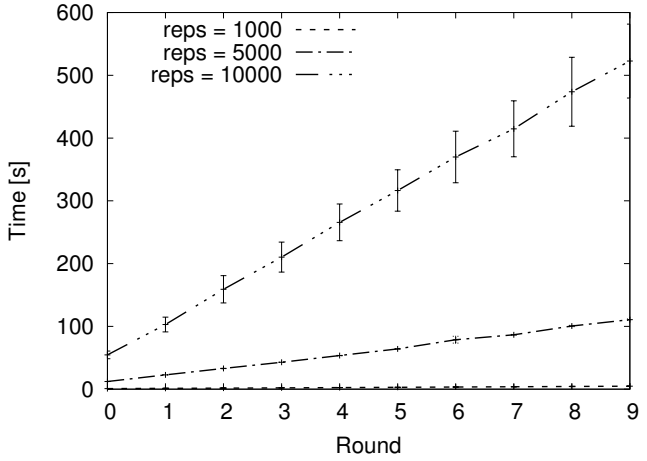

Figure 7. Retrieval scalability for MongoDB ($reps$ records are added, and $reps$ random records are retrieved in each round).

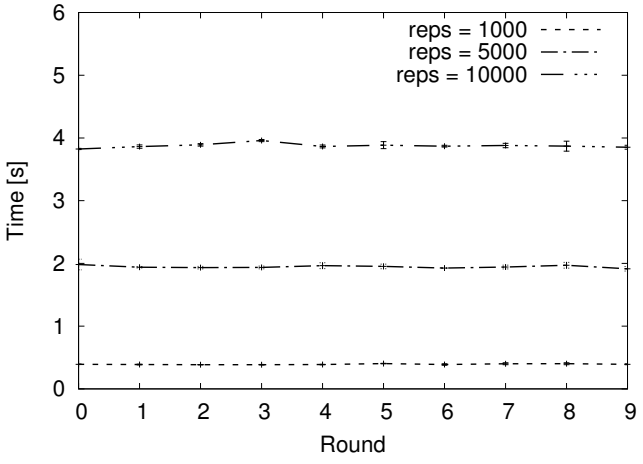

Figure 9. Retrieval scalability for MySQL ($reps$ new records are added, and $reps$ random records are retrieved in each round).

that the performance of MongoDB is dramatically decreasing with the increasing number of records in the store. Also, the absolute values achieved by MongoDB are not very good, to retrieve 10000 random annotations from a database with 90000 documents, almost eight minutes are required.

The retrieval times for the same amount of data from the neo4j database of the same size are much lower as can be seen in Figure 8 (please note that the y axis was scaled comparing to Figure 7). Also, the scalability of neo4j is much better, neo4j produces constant answer times regardless of the size of the database. For comparison with the MongoDB, time to retrieve 10000 random entities from a neo4j graph (regardless of its size), varies around $1.37s$. When comparing neo4j with MySQL (i.e., Figure 8 with Figure 9), one can see that neo4j is faster across the parameter range by roughly a factor of 3, but MySQL remains much faster than MongoDB.

B. Storing scalability

The situation is a little bit different for creation times. We depicted them in Figure 10 and Figure 11. This time MongoDB outperforms its competitors. Both neo4j and MySQL suffer under high variance in the query times. For higher value of $reps = 10000$, neo4j is clearly the slowest in creating new annotations. We believe that the main reason for this is the fact that neo4j is using the most sophisticated model for annotations. A creation of an annotation requires creation of two nodes in the graph (one for target and one for body) and an edge connecting them. Upon creation, neo4j is also verifying the uniqueness constrain for the created nodes. MySQL is storing the values for annotation in one table, and MongoDB does not verify the uniqueness. As stated above, in a more realistic scenarios, one would have to use multiple tables for storing annotations in MySQL, and run transactions to add new values. Also in case of MongoDB, firstly a laborious search across the database would be required to avoid duplicates. It would increase the query times by the amount depicted in Figure 7.

VII. Discussion

Our evaluation included three distinct classes of products: document store (MongoDB), relational database (MySQL),

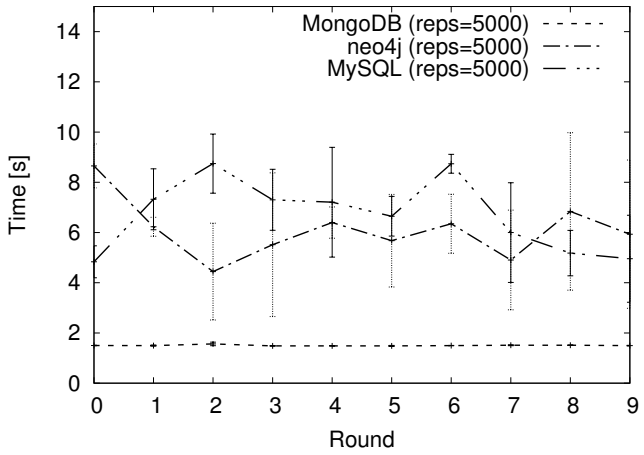

Figure 10. Comparison of creation scalability ($reps = 5000$ new records are added in each round).

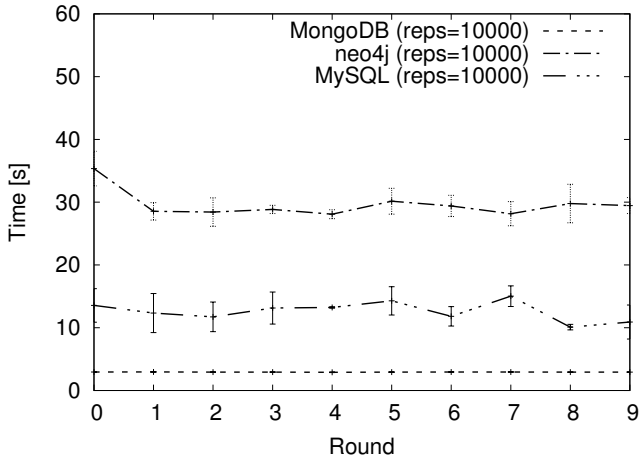

Figure 11. Comparison of creation scalability ($reps = 10000$ new records are added in each round).

and a native graph database (neo4j). The evaluation can be roughly split in two main experiments: storing and retrieving. In terms of retrieving of annotations the neo4j is a clear winner, followed by MySQL. The storing part was best handled by MongoDB, which displayed very good scalability. On the overall, we have two products that excel in one metric: neo4j in retrieval and MongoDB in storing. MySQL occupies the middle position in both dimensions.

We discussed the suitability of the products. We argued that neo4j might be best suited also for handling more sophisticated use cases than those in our tests. On the other hand, the results for MySQL would be probably degraded in case of more complicated (and thus more join-intensive) queries.

The final recommendation with respect to kind of storage that should be used to manage semantic annotations depends on the ratio between reads and writes the service has to handle. In most of the cases the reading would be the most dominant operation and then probably neo4j is the best option. In case of very write-intensive applications, MongoDB might be a better option. Also more complex deployments with both MongoDB and neo4j supporting respectively write and read operations (and synchronizing the storage in the background) are conceivable.

The evaluation was conducted with the help of the above described framework. It is hard to quantify how easy it is to use, but the actual deployment of the software and starting of experiments was done with help of Docker and took very little time and effort. We stick to our definition of reproducibility (see Section I) and exclude the influence of the hardware. We can, however, see that it has some influence on the results. A clear indicator for that is the variance of the measured values across test runs. We include the variance in the evaluation plots. The goal of the reproducible evaluation is not to obtain exactly the same curves on the plot, but rather quantitatively same results within given range of confidence intervals.

VIII. Conclusion and Future Work

In this paper, we have evaluated different options for storing semantic annotations. From these options native graph database seems to be the best general candidate technology, some specific use cases can be better off with other tested solutions.

Our main contribution is the framework for conducting the above described experiments in a reproducible fashion. This framework is based on a novel technology and allows a seamless exchange of not only code used for benchmarks but complete, ready-to-run experimental setups between the researchers. We implemented the framework in such a way that it is possible to reproduce our experiments, process the results, and produce plots by issuing just one command. Furthermore, by sharing both the source code of our testing scripts and formal Docker-based description of their installation, we allow researchers to reuse and adjust them to answer different research questions. We discussed how future-proof the framework is and, although, such predictions are always hard, we argued that the framework has some potential to better understand and repeat our experiments even in the future.

In our future work, it would be interesting to follow two directions. Firstly, incorporate more sophisticated benchmarks in our framework to validate our current recommendations with respect to the tested technologies. Such benchmarks could bring to surface currently hidden scalability and redundancy management problems we alluded to. Secondly, the application of the framework to other use cases and other technologies could cast some light on its extensibility. Such more extensive applications would also help in defining more robust ways of exchanging data between the steps of the workflow.

Acknowledgment

The work has been supported by EUDAT2020, funded by the European Union under the Horizon 2020 programme - DG CONNECT e-Infrastructures (Contract No. 654065).

References

[1] J. Rybicki and B. von St. Vieth, “Reproducible evaluation of semantic storage options,” in Proceedings of the 3rd IARIA International Conference on Big Data, Small Data, Linked Data and Open Data (ALLDATA ’17), Apr. 2017, pp. 26–29, ISBN: 978-1-61208-552-4, ISSN: 2519-8386.

[2] W. Gentzsch, D. Lecarpentier, and P. Wittenburg, "Big data in science and the EUDAT project," in Proceedings of the Service Research and Innovation Institute Global Conference, Apr. 2014, pp. 191–194, ISBN: 978-1-4799-5193-2, ISSN: 2166-0786.

[3] W3C Web Annotation Data Model. [Online]. Available: https://www.w3.org/TR/annotation-model/ [retrieved: Nov., 2017]

[4] A JSON-based Serialization for Linked Data. [Online]. Available: https://json-ld.org/ [retrieved: Nov., 2017]

[5] MongoDB. [Online]. Available: https://www.mongodb.com/ [retrieved: Nov., 2017]

[6] Apache CouchDB. [Online]. Available: https://couchdb.apache.org/ [retrieved: Nov., 2017]

[7] C. Gormley and Z. Tong, Elasticsearch: The Definitive Guide. O'Reilly Media, 2015, ISBN: 978-1-449-35854-9.

[8] J. Webber, "A programmatic introduction to Neo4j," in Proceedings of the 3rd ACM Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12), Oct. 2012, pp. 217–218, ISBN: 978-1-4503-1563-0.

[9] MySQL. [Online]. Available: https://www.mysql.com/ [retrieved: Nov., 2017]

[10] M. B. Gurstein, "Open data: Empowering the empowered or effective data use for everyone?" First Monday, vol. 16, no. 2, 2011, ISSN: 13960466.

[11] J. Rybicki and B. v. St. Vieth, "DARIAH Meta Hosting: Sharing software in a distributed infrastructure," in Proceedings of the 38th IEEE International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO '15), May 2015, pp. 217–222, ISBN: 978-9-5323-3082-3.

[12] Docker. [Online]. Available: https://www.docker.com/ [retrieved: Nov., 2017]

[13] Docker Compose. [Online]. Available: https://docs.docker.com/-compose/ [retrieved: Nov., 2017]

[14] D. De Witte, L. De Vocht, R. Verborgh, K. Knecht, F. Pattyn, H. Constandt, E. Mannens, and R. Van de Walle, "Big linked data ETL benchmark on cloud commodity hardware," in Proceedings of the ACM International Workshop on Semantic Big Data (SBD '16), 2016, pp. 1–6, ISBN: 978-1-4503-4299-5.

[15] Amazon Web Services. [Online]. Available: https://aws.amazon.com/ [retrieved: Nov., 2017]

[16] Project Jupyter. [Online]. Available: http://jupyter.org/ [retrieved: Nov., 2017]

[17] A. Pacaci, A. Zhou, J. Lin, and M. T. Özsu, "Do we need specialized graph databases?: Benchmarking real-time social networking applications," in Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems (GRADES '17), 2017, pp. 1–7, ISBN: 978-1-4503-5038-9.

[18] S. Riggs, G. Ciolli, and G. Bartolini, PostgreSQL Administration Cookbook, 9.5/9.6. Packt Publishing, 2017, ISBN: 978-1-785-88318-7.

[19] D. Hernández, A. Hogan, C. Riveros, C. Rojas, and E. Zerega, "Querying wikidata: Comparing SPARQL, relational and graph databases," in Proceedings of the 15th International Semantic Web Conference (ISWC '16), Oct. 2016, pp. 88–103, ISBN: 978-3-319-46546-3.

[20] Wikidata. [Online]. Available: https://www.wikidata.org/ [retrieved: Nov., 2017]

[21] Linked Data Benchmark Council (LDBC). [Online]. Available: http://www.ldbcouncil.org/ [retrieved: Nov., 2017]

[22] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz, "The LDBC social network benchmark: Interactive workload," in Proceedings of the ACM International Conference on Management of Data (SIGMOD '15), 2015, pp. 619–630, ISBN: 978-1-4503-2758-9.

[23] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardto, H. Chafio, M. Capotă, N. Sundaram, M. Anderson, I. G. Tănase, Y. Xia, L. Nai, and P. Boncz, "LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms," Proceedings of the Very Large Data Base Endowment, vol. 9, no. 13, Sep. 2016, pp. 1317–1328.

[24] D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram, and V. Stodden, "Reproducible research in computational harmonic analysis," Computing in Science & Engineering, vol. 11, no. 1, 2009, pp. 8–18, ISSN: 1521-9615.

[25] J. Freire, P. Bonnet, and D. Shasha, "Computational reproducibility: State-of-the-art, challenges, and database research opportunities," in Proceedings of the ACM International Conference on Management of Data (SIGMOD '12), 2012, pp. 593–596, ISBN: 978-1-4503-1247-9.

[26] S. Eglen, B. Marwick, Y. Halchenko, M. Hanke, S. Sufi, P. Gleeson, R. A. Silver, A. Davison, L. Lanyon, M. Abrams et al., "Towards standard practices for sharing computer code and programs in neuroscience," Nature Neuroscience, vol. 20, no. 6, 2017, pp. 770–773, ISSN: 1097-6256.

[27] K. M. Chandy, J. Kiniry, A. Rifkin, and D. Zimmerman, "Webs of archived distributed computations for asynchronous collaboration," The Journal of Supercomputing, vol. 11, no. 2, 1997, pp. 101–118, ISSN: 1573-0484.

[28] R. Kahn and R. Wilensky, "A framework for distributed digital object services," International Journal on Digital Libraries, vol. 6, no. 2, Apr. 2006, pp. 115–123, ISSN: 1432-5012.

[29] ePIC Consortium. ePIC – persistent identifiers for eResearch. [Online]. Available: http://www.pidconsortium.eu/ [retrieved: Nov., 2017]

[30] Docker Hub. [Online]. Available: https://hub.docker.com/ [retrieved: Nov., 2017]

[31] Official Docker MongoDB Image. [Online]. Available: https://hub.-docker.com/_/mongo/ [retrieved: Nov., 2017]

[32] Official Docker neo4j Image. [Online]. Available: https://hub.docker.-com/_/neo4j/ [retrieved: Nov., 2017]

[33] Official Docker MySQL Image. [Online]. Available: https://hub.-docker.com/_/mysql/ [retrieved: Nov., 2017]

[34] J. Rybicki. Annotations scalablity: Source code reposiotry. [Online]. Available: https://github.com/httpPrincess/annotations-scalability [retrieved: Nov., 2017]

[35] Gnuplot. [Online]. Available: http://gnuplot.sourceforge.net/ [retrieved: Nov., 2017]

[36] J. Rybicki. Docker image for Tester. [Online]. Available: https://hub.docker.com/r/httpprincess/tester/ [retrieved: Nov., 2017]

[37] ——. Docker image for Results Processor. [Online]. Available: https://hub.docker.com/r/httpprincess/processor/ [retrieved: Nov., 2017]

[38] ——. Docker image for Orchestrator. [Online]. Available: https://hub.docker.com/r/httpprincess/orchestrator/ [retrieved: Nov., 2017]

[39] ——. Docker image for Results Visualizer. [Online]. Available: https://hub.docker.com/r/httpprincess/orchestrator/ [retrieved: Nov., 2017]