The GeoRDFBench Framework: Geospatial Semantic Benchmarking Simplified

Theofilos Ioannidis[1 [0000−0002−1754−8748]] and Manolis Koubarakis[1 [0000−0002−1954−8338]]

Department of Informatics and Telecommunications
National and Kapodistrian University of Athens, Greece
{tioannid,koubarak}@di.uoa.gr

Abstract. We present the GeoRDFBench framework, whose purpose is to assist and streamline the researcher's work in the field of benchmarking geospatial semantic stores. The runtime API models all identified benchmark components, groups them, forms specialization hierarchies of classes and interfaces for them, and supports serialization to and deserialization from external JSON specification files. The increased reusability of these JSON benchmark specifications, allows focus to stay on the research task ahead while minimizing the time from idea conception to benchmark results and useful conclusions. Geospatial RDF store architecture and behavior is unified by generalizing the repository and connection functionalities of the three most common RDF framework APIs used by RDF stores: OpenRDF Sesame, Eclipse RDF4J and Apache Jena. GeoRDFBench goes even further and models the application and database server modules present in some stores and automates their life-cycle management during experiment execution. The framework comes with several geospatial RDF stores, implemented as separate runtime-dependent modules. Each module contains scripts for repository generation and experiment execution, which allows for a quick start on using the platform. RDF modules include: RDF4J, GraphDB, Stardog, Strabon, OpenLink Virtuoso and Jena GeoSPARQL.

Keywords: Geospatial · Semantic · Benchmarking · Framework.

1 Introduction

Many projects have shown[1] [2] [3] that spatial and temporal aspects of Linked Open Data (LOD) are as important and critical, as sensitive thematic information, in order to guide decision making [22]. Graph database systems with varying degrees of spatial support have recently been used to manage very large LOD datasets [10,21,4,2,28,15,19,7,16]. Selecting the most suitable system for user

[1] INSPIRE Directive https://inspire.ec.europa.eu/inspire-directive/2
[2] SWING https://cordis.europa.eu/project/id/026514
[3] ExtremeEarth project https://earthanalytics.eu/index.html

needs and budget requires *frequent evaluation of available systems, on infrastructure within the defined budget, with desired queryloads run against datasets of appropriate size and content.*

Benchmarks proposed in the literature have focused on automating some of the arduous and repetitive tasks of benchmarking. Parametric ontology-based synthetic generators [13,5,24,12,9] have assisted in creating datasets of desired size and attributes, while log-mining techniques [24,32,9] helped creating more application specific querysets. Benchmarking cloud platforms featuring distributed file systems, containerization technologies and intuitive web UIs have allowed reuse of implemented systems and workloads and ease of management. In all cases however, the benchmark researcher has not been spared the effort to deal with system configuration and optimization, spatial indexing setup, detailed query execution exception handling and learning required technology stacks and platform APIs.

Motivation for this work. Our experience with geospatial benchmarks on single node and Spark-based distributed RDF stores along with synthetic data and query generation [15,4,12] has led us to believe that the geospatial semantic store research area would greatly benefit by the introduction *of a lean but extensible geospatial benchmarking framework that aims to assist system evaluators in creating new customizable benchmarks with many different systems, many workload types, in as fast, credible and repeatable way as possible.*

The proposed GeoRDFBench framework's suffix "Bench" does not stand for benchmarking only, but also as a reminder that our intention is for it to be used as the *"garage bench"* were a researcher will find the necessary tool set to try quickly and safely new ideas and get results. Our work does not place emphasis on some of the nice to have features, such as UI, or containerized[4] module execution and focuses initially on benchmarking single node geospatial graph stores through the console. It allows parallel experiment execution of implemented stores in the same node. Its architecture however allows its installation in clustered environments with minimal additions, to also support distributed file access API.

To the best of our knowledge, there is no similar work that combines a substantial part of the following features that are also the core contributions of our framework:

1. It abstracts and implements specification hierarchies for components required to setup and run an experiment on a geospatial RDF store. Components include: *datasets, querysets, experiment execution, workloads, logging specifications, report sinks, and hosts.*
2. It abstracts and generalizes repository/connection functionality, from the three best known RDF framework APIs: *(i) OpenRDF Sesame, (ii) Eclipse RDF4J and (iii) Apache Jena.*

[4] GeoRDFBench's site includes containerized images for demonstration purposes.

3. It abstracts and implements class hierarchies of geospatial enabled RDF stores according to their basic module architecture and the RDF framework APIs they support.
4. It provides several implemented modules[5] for representative geospatial RDF stores of different architectures and which use different RDF framework APIs: *(i) RDF4J, (ii) GraphDB, (iii) Stardog, (iv) Strabon, (v) Virtuoso, (vi) Jena GeoSPARQL*. These modules can act as *templates* for other stores with similar architecture and in most cases the required code is minimal.
5. It provides an *external library of JSON serialized instances* for datasets, querysets, experiment execution, workloads specifications of a geospatial benchmark, a PostgreSQL implementation of the JDBC report sink and hosts with Linux operating system.
6. It features a *single experiment execution loop* independent of the store architecture and RDF framework API used.
7. It enables *automatic result accuracy verification* for workloads that have embedded an expected resultset, while query execution accuracy results are persisted with other statistics.
8. It allows *synchronous or deferred persistence of experiment results with customized statistics* to a user-defined report sink.
9. Since GeoSPARQL is a superset of SPARQL, the framework can also be used, as is, for SPARQL benchmarks.

The organization of the rest of the paper is as follows. Section 2 discusses related work. Section 3 presents the high level architecture of the framework while section 4 focuses on GeoRDFBench's runtime. Finally, section 5 presents conclusions and future work.

2 Related Work

In this section, we present related work on graph and geospatial graph store categories, architecture, evaluation criteria and benchmarking.

Graph Store Categories. Graph stores in general, follow either the RDF or the LPG approach. RDF as a data model has good expressivity, while featuring a standardized declarative query language SPARQL[6] and a standardized spatial vocabulary GeoSPARQL [23]. LPGs, on the other hand, excel in graph traversal and path search for analytics and machine learning. But they lack standardization as there are several data models and languages from high-profile vendors and institutions, such as, Neo4j's Cypher[7], Apache TinkerPop Gremlin [31], the Oracle supported PGQL [29] and G-Core [1] from the Linked Data Benchmark Council (LDBC). Another issue is that some of these languages are declarative while others are procedural. To conclude, at the time of writing of this paper, most geospatial graph databases that support complex geometries support the RDF model.

[5] GeoRDFBench framework is a Java Maven multi-module POM project.
[6] https://www.w3.org/TR/sparql11-query/
[7] https://neo4j.com/developer/cypher/

Geospatial Graph Store Architecture. All stores have some front-end application, usually a terminal or web-based `console`, through which the user can create repositories, load datasets to and run queries against them. Depending on the system, this console may communicate directly with the back-end or may relay requests to an application module acting as a proxy.

The `application module` is usually an HTTPS server or servlet container, such as, Nginx or Eclipse Jetty, with multi-user, load balancing and connection reuse as general capabilities. It also allows centralized semantic store configuration with sane default values for all unconfigured user sessions.

Due to the large size of linked datasets, many systems, apart from the console embedded ingestion utilities, they offer dedicated `bulk loading utilities` which can speed up the import step. Such an example is the Preload and LoadRDF tools in Ontotext's GraphDB. These tools, usually follow the pattern of, deferring index creation on one hand while using multiple executing threads of contemporary CPUs to ingest a hint-driven chunk-dissected dataset.

For the `back-end module`, storage type and indexing methods are the usual differentiating factors. Some stores, mostly research-oriented and especially early ones, employ rigid architectures based on specific implementation recipes. For example, Parliament [2] uses the embedded key-value database Berkeley DB with a standard R-tree spatial index, Strabon [21] uses PostgreSQL and PostGIS with an R-tree over GiST [14] spatial index, uSeekM follows the same path but for spatial information only and native file storage is used for storing and managing thematic information with B+ trees, while Oracle Spatial And Graph[8] uses an R-tree spatial index on top of its proprietary industry leading RDBMS solution. More contemporary market-driven stores offer elastic architectures which effectively decouple modules from specific implementation choices by offering many compatible alternatives for each one of them. For example, Virtuoso offers virtual graphs over many well known data and file formats such as Excel, XML files and RDBMS data sources.

Geospatial Graph Store Evaluation Criteria. The growth rate of LOD sizes questions the ability of graph databases to persist this big data, while at the same time pose a critical challenge for the performance of these stores under query loads of interest. For spatial data, we face an additional challenge which is the approximate nature of data representation, especially with the indexing process. Most spatial indexing algorithms, such as, quad [11] and geohash[9] prefix trees support a precision parameter which basically controls how many results will match a spatial filter. Better accuracy requires more storage and brings a performance penalty, so most systems try to balance between the two. The spatial index algorithm and precision are either fixed for the store, defined upon database creation or dynamically defined even after database creation. Setting up a system with lower accuracy, has the benefit of reduced storage size, bulk load

[8] https://docs.oracle.com/en/database/oracle/oracle-database/19/spatial-and-graph.html

[9] https://en.wikipedia.org/wiki/Geohash

time and query execution times. Therefore, we have 3 important check points that a geospatial semantic benchmark should measure when testing spatially enabled stores: (i) *bulk load ability* for huge dataset sizes, (ii) *query execution performance* for various query loads and (iii) *query execution accuracy*. While a valid accuracy test is comparing the query resultset against the expected resultset, storage requirements for large expected resultsets make this impractical as a general approach. Low selectivity queries against a 100M-triple dataset or even highly selective queries against a 10G-triple dataset can yield 10M-triple resultsets. A more general approach is to evaluate the system accuracy in a piecemeal fashion using a benchmark comprising several workloads. Small realworld or synthetic datasets with simple and highly selective queries that use each one of the operators of interest, make it easy to check accuracy and find implementation or configuration issues with a system. With the previous issues resolved[10], we proceed with large realworld datasets with querysets of interest where for each query we compare the number of returned results against the expected number of results. Under the preconditions mentioned, this is a good indicator of the query accuracy since it is fast to verify and with low storage requirements which makes it easy to persist and disseminate.

Benchmarking Graph Stores. Various SPARQL and GeoSPARQL benchmarks have been devised over the past 20 years to test the supported features and performance of graph stores. Well known SPARQL benchmarks include: LUBM [13], BSBM [5], DBpedia SPARQL benchmark (DBPSB) [24] and the Social Network Benchmark (SNB) [9], just to mention a few. GeoSPARQL benchmarks have been presented in [27,26], the benchmark Geographica in [12], a smart city services related benchmark in [3], a compliance benchmark in [18] and Geographica 2 in [15].

Benchmarking is a notoriously *difficult, time-consuming, resource intensive, high complexity, multi-parameter and error prone process* even when human nature's bias is not present to favor one of the proposed solutions. Since graph stores are continuously evolving and offer improved efficiency and new capabilities, *it is also a process that needs to be repeated regularly*, if the benchmark results are to reflect a valid image of the graph store ecosystem.

Benchmarking Frameworks. A *benchmarking framework* is a software platform that allows: (i) easy integration of systems of interest, (ii) easy integration of existing benchmarks, (iii) easy generation and customization of benchmark datasets and querysets, (iv) running experiments of a benchmark against one or more systems, (v) collecting experiments results and system logs, (vi) result analysis and finally (vii) easy experiment verification. Some of these features appeared as new ideas or automations included in different benchmarks, which however *should not create the impression that these benchmarks can be considered proper frameworks.*

In particular, several SPARQL and GeoSPARQL benchmarks have generalized or automated the queryset and dataset generation task. For example,

[10] Removing problematic queries or reconfiguring the system.

LUBM, which focuses on reasoning, features a university ontology-based synthetic data generator able to scale to arbitrary sizes. DBPSB's queryset creation process is based on querylog mining, clustering and SPARQL feature analysis, which is applied to the DBpedia knowledge base and shows that performance of triple stores is by far less homogeneous than suggested by non application-specific benchmarks. FEASIBLE [32] suggests an automatic approach for the generation of application-specific benchmark querysets (SELECT, ASK, DESCRIBE and CONSTRUCT) out of the application's query logs history, thus enhancing insights as to the real performance of triple stores employed for a given application. IGUANA [6] innovates by providing an execution environment which can measure the performance of RDF stores during data loading, data updates as well as under different loads and parallel requests. LITMUS [33] proposes uniform benchmarking of non-spatial data management systems supporting different query languages, such as, SPARQL and Gremlin. Geographica and Geographica 2 include an ontology-based geospatial synthetic generator able to create spatial datasets of arbitrary size and also generate the corresponding queryset with a user defined thematic and spatial selectivity. Kobe [20] cloud benchmarking engine for federated query processors includes the GeoFedBench [34] benchmark, which focuses on validation of the actual crop land usage against the Austrian land survey dataset.

A more recent idea is that of a *benchmarking framework platform*. These are benchmarking frameworks designed for deployment to cloud infrastructures, with distributed file systems and containerization technologies. They are multi-user environments where researchers can store and share datasets, querysets, execution results and system modules. HOBBIT [30], the most complete of these platforms, extends the scope of benchmarking to the entire linked data life-cycle [25], such as *link discovery* [17], employs intuitive web UIs and allows the integration of systems in various programming languages. Overall, it seems that HOBBIT achieves generality to accommodate benchmarks across the whole Linked Data life-cycle, achieves component flexibility with containerization, promotes language independence, vertical scalability and compliance to FAIR initiative. On the other hand, HOBBIT increases platform complexity, sacrifices usability for new users and does not provide out-of-the-box benchmark-specific and system-specific knowledge reusability for benchmark researchers. Human scholars need to heavily invest on this framework and still not get the expected assistance for their effort. The HOBBIT platform and FAIR Data Principles are further discussed in the GeoRDFBench Framework site[11].

3 GeoRDFBench: A Framework Simplifying Geospatial Semantic Benchmarking

In this and the following section we present the technical details of the GeoRDFBench framework, starting with its high level architecture which is shown in Figure 1.

[11] https://geordfbench.di.uoa.gr

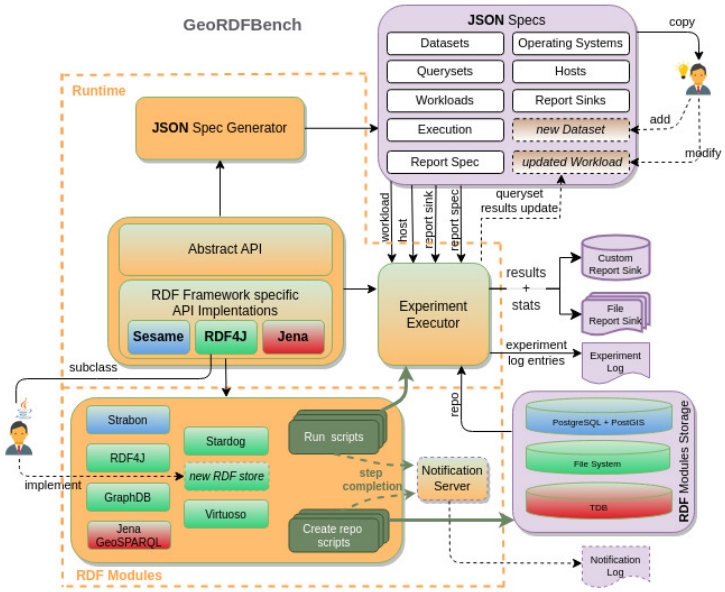

Fig. 1: Architectural overview of GeoRDFBench Framework

The system consists of two main parts: the *runtime* and the *RDF modules*, depicted inside the dashed border. The runtime is the engine and fabric of the framework and it is responsible for generating default JSON benchmark specifications and executing experiments initiated by the RDF modules' run scripts. The RDF modules is where pre-implemented and newly implemented RDF stores reside that can participate in experiments. Stores use their *repository creation script* to create repositories and import data to them. Each store's *experiment run script* initiates a benchmark experiment and eventually invokes the runtime's *experiment executor* component passing all required inputs which include, among others: the RDF store, the repository, the benchmark workload, the host where the experiment is conducted on and the report sink where experiment results and statistics will be stored. Both types of scripts send progress messages to the optionally enabled, remote or local, *notification server* which logs them and serves as a useful non-intrusive monitoring tool for the researcher.

The default *JSON specifications* correspond to the Geographica 2 benchmark's components and are generated by the *JSON Specs Generator*. This "starter dough" library is stored on the file system separately from GeoRDFBench and can be easily copied or modified by the user. The workload specifications in particular can be modified by injecting to them an experiment's resultset[12], which allows for accuracy validation of future experiments with the same workload.

[12] Number of results for each query.

4 GeoRDFBench Runtime: The Framework Engine

The GeoRDFBench runtime consists of four components: (i) the `Abstract API`, (ii) the `RDF Framework Specific API Implementations`, (iii) the `Experiment Executor` and (iv) the `JSON Specification Generator`.

In this section, the term *system* is used to refer to the repository functionality of a geospatial RDF store, while the term *system under test (SUT)* is used to refer to the system-host pair, along with management capabilities of the system's application and database server status, if they are present. The SUT knows how and in which sequence to start and stop the application server, repository and database server of the system and to clear system caches. SUT is also the vehicle with which experiment timed queries are executed.

4.1 Abstract API

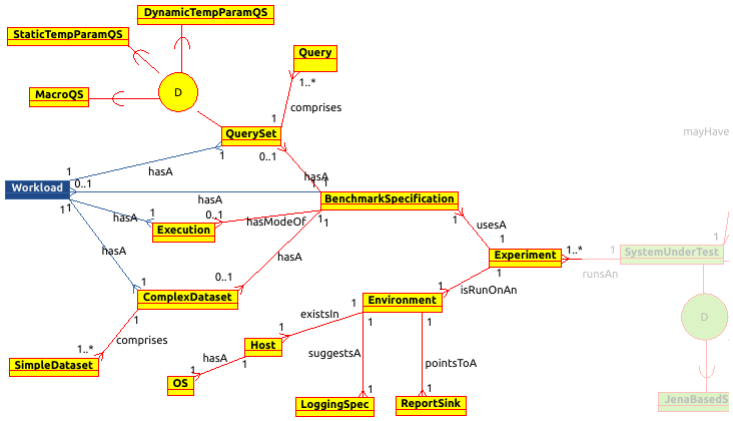

Fig. 2: EER diagram of Abstract API - Experiment Components

This is a core part of the GeoRDFBench's benchmarking API. It abstracts the properties, functionalities and interactions of the *benchmark experiment components* and the *SUT*. On the conceptual level, the two simplified[13] Enhanced-ER (EER) diagrams, in Figure 2 and in a part of Figure 3, show the important entities, their specializations, how they are associated, along with the corresponding structural constraints (cardinality constraints) for these associations.

On the implementation level, the *Abstract API* creates class hierarchies for each component type, exposes common functionality with appropriate interfaces and uses abstract classes to pull up properties and provide default implementations for operations.

[13] Only important entities, specializations and relationships are depicted while attributes are omitted.

Experiment Components Figure 2 depicts the experiment components which are detailed below:

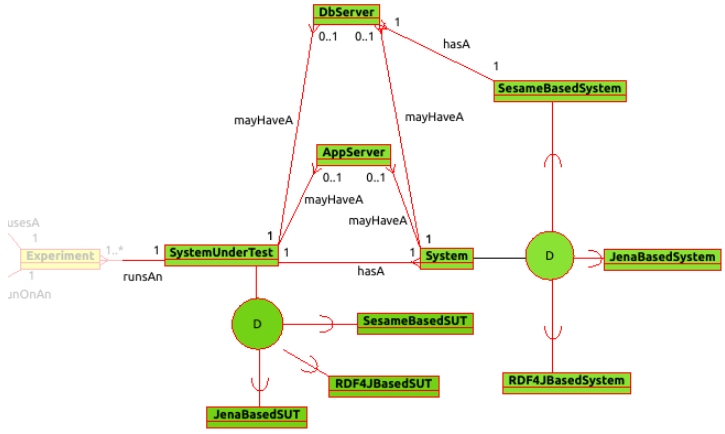

Fig. 3: EER diagram of Abstract API - System Under Test

Benchmark Specification. This group includes all components necessary to describe a benchmark which are independent of the platform where the experiment runs. There are two forms available: (i) the *detailed form* uses independent component specifications and (ii) the *compact form* which uses the *workload* "container" concept to group several components. With the exception of the *workload*, all components described below are part of the *detailed form*.

1. **dataset**: the entity `ComplexDataset`, in Figure 2, represents the "complex" or "composite" geospatial dataset which can comprise one or more "simple" geospatial datasets represented by the `SimpleDataset` entity. The *simple dataset* specification contains: a logical name, the dataset relative path location, the filename, the RDF serialization format, a map of dataset relevant namespace prefixes, a map of properties that link features with their geometries which represent their spatial extent e.g., `geo:hasGeometry`, a map of properties that link a geometric element with its WKT serialization e.g., `geo:asWKT`, and the scaling factor used in case of a synthetic dataset. The *complex dataset* specification contains: a logical name, the dataset base path location, a map of contexts (named graphs) and the list of simple datasets comprising it. Experiments use only complex datasets. The runtime computes the final location of a complex dataset's files by concatenating the host's base path for dataset files, the complex dataset's base path, the simple dataset's relative path and filename.
2. **queryset**: the entity `QuerySet`, in Figure 2, represents the geospatial queryset which can comprise one or more queries represented by the `Query` entity.

The *query* specification contains: a label, the GeoSPARQL query text which may contain replaceable tokens (template parameters), a flag that signals the existence of spatial predicate and the accuracy validation indicator. The accuracy indicator denotes whether the expected number of results returned by the query is dataset-dependent, template-dependent, or independent. The *queryset* specification contains: a logical name, the queryset path location, a map of queryset relevant namespace prefixes, a map of fixed, static template, or dynamic template queries with arithmetic index and replacement maps that assist in creating ground queries from template queries. Experiments use only querysets, which can be filtered at run time. The `StaticTempParamQS` subclass models sets of template parameter queries which have fixed parameter values for all queries. `DynamicTempParamQS` subclass models sets of template parameter queries which may have different value for each parameter for each query. While these subclasses are useful for "micro"[14] experiment types, the `MacroQS` subclass and its specializations[15] are useful for "macro"[16] experiment types.

3. **execution specification**: the entity `Execution`, in Figure 2, describes which experiment action *(**run**, print)* to take, the query execution types *(cold, warm, cold_continuous)* and number of repetitions per type, the query repetition timeout and total timeout for all repetitions, the delay period for synchronous clearing of system caches, the function to use for aggregating execution times and the policy to follow when a cold execution times out. The non-default *print* action triggers a pseudo-execution which generates the ground queryset for inspection purposes.
4. **workload**(compact form): the entity `Workload`, in Figure 2, represents the compact form of the benchmark experiment description: *dataset + queryset + execution specification*. The execution specification, however, depends almost entirely on the queryset at hand, since it describes the experiment logic to apply for the queryset in order for it to achieve its purpose. Therefore, the exact internal representation of the workload is:

```
workload{ dataset{ ..}, queryset{ .., execution specification{ ..} } }.
```

Experiment environment. This group includes all platform dependent components, such as, what hardware platform and operating system the experiment runs on and which storage facility the results and statistics are to be recorded to. The result report store does not need to reside in the experiment execution platform.

1. **operating system**: the entity `OS`, in Figure 2, represents the host's operating system and features a name, the shell command path, the commands for synchronizing cached data to persistent storage and the one for fully clearing caches (pagecache, dentries and inode).

[14] Independent queries, each run several times with cold or warm caches.
[15] Not shown in Figure 2 for simplicity reasons.
[16] A sequence of queries representing a case scenario, that is run repeatedly as a whole.

2. **host**: the entity `Host`, in Figure 2, represents the hardware platform where the benchmark experiment is taking place. It has the host name, IP address, total RAM (GBs), the base path for dataset files, the base path for RDF store repository files and the base path for the default reports and statistics. Since, the operating system specification depends entirely on the host on which it is installed, the exact internal representation is:

```
host{ .., operating system{ ..} }
```

3. **report sink**: the entity `ReportSink`, in Figure 2, describes the experiment result report store, where the customized reports and statistics will be sent. The default report store is a PostgreSQL JDBC implementation and has as properties, the driver name, hostname, alternate hostname, port, database name, user and password. Alternate hostname allows for having a fall-back database where results from extremely long running experiments can be saved. The PostgreSQL report store has as default behavior the *deferred insertions* for query execution results, that involves an experiment result collector which flushes results upon experiment termination. In this way, we avoid synchronous result insertions to the report sink, which would be disrupted by the repetitive restarts of the database component for SUTs using the same DBMS as the report sink. The target report sink database schema is generated with the help of the runtime-bundled *database generation SQL script*.
4. **logging specification**[17]: the entity `LoggingSpec`, in Figure 2, allows customization of the number of resultset entries to be logged during the query execution scanning phase of the `Experiment Executor`. A positive non zero integer value allows for a sample of the results returned by each query to be recorded in the experiment log and can be used as a proof of concept that a system performs accurately or similar to other systems. Such a setting is useful in early benchmarking phases and can help identify, early on, issues with disabled plugins, external libraries, or with incorrect results by non-compliant function behavior. A zero value, on the other hand, allows for very accurate calculation of the query response time and is useful in the final benchmarking phase.

Systems and SUTs In Figure 3, the parent concepts of system and SUT components only, are also part of the Abstract API. The entity *System* represents an RDF framework independent geospatial semantic store and more specifically the repository aspect of it. It is described by a map of properties and their values, such as repository location and name, system relevant namespace prefixes, as well as various indexing parameters. It also has a connection property which allows query execution and a flag to denote whether the store has been initialized. The entity *SystemUnderTest* on the other hand represents the combination of

[17] Also mentioned as ReportSpec in the framework

a *System* with its optional application and database server components, represented by *AppServer* and *DbServer* respectively.

On the implementation level, the Abstract API comprises two layers: (i) the *Geospatial System Abstraction Layer*, which is depicted in the lower left two hierarchy levels of Figure 4, and (ii) the *System Under Test (SUT) Abstraction Layer*, which is the lower right level of the same figure, both of which are explained below. Due to the fact that GeoRDFBench was influenced by the Geographica 2 benchmark, several implementation components have "Geographica" in their name, which for all intents and purposes can be interpreted as "Geospatial".

1. **Geospatial System Abstraction Layer** This layer comprises one interface that describes a geospatial RDF store and one abstract class that implements the RDF framework independent common functionality.
   (a) The *Geospatial Graph System Interface* (`IGeographicaSystem`), is a contract that requires functions for: setting a map of system properties, system initialization, system termination and a function that returns system specific namespace prefix mappings.
   (b) The *Base Abstract Implementation* (`AbstractGeographicaSystem`) of `IGeographicaSystem`, is an abstract class that uses generics and encapsulates the system properties map, the initialization status, the generic repository "connection", which is RDF Framework specific and the skeleton functionality to handle these. This generic "connection" corresponds to an appropriate RDF Framework abstraction that allows creating query instances on a system repository.
2. **System Under Test (SUT) Abstraction Layer** This layer comprises the generic *Geospatial Graph SUT Interface* (`ISUT`), which is a contract that requires functions for: retrieving the host, the generic "system", execution and report specifications, starting and terminating the application and database server, making system dependent translations of the queryset and executing timed queries.

4.2 RDF Framework Specific APIs

The second part of the core API, on the conceptual level, is depicted in part of Figure 3 which includes the specializations of system and SUT. In a similar manner, system and SUT concepts have three child entities to model the corresponding three RDF framework specific concepts: *RDF4JBasedSystem*, *JenaBasedSystem*, *SesameBasedSystem*, *RDF4JBasedSUT*, *JenaBasedSUT* and *SesameBasedSUT*.

On the implementation level, this part comprises two parts: (i) the *RDF Framework Specific System Layer*, which is depicted by the left side of "RDF Framework Implementation" level of Figure 4, and (ii) the *RDF Framework Specific SUT Layer*, which is the right side of the same level, both of which are explained below.

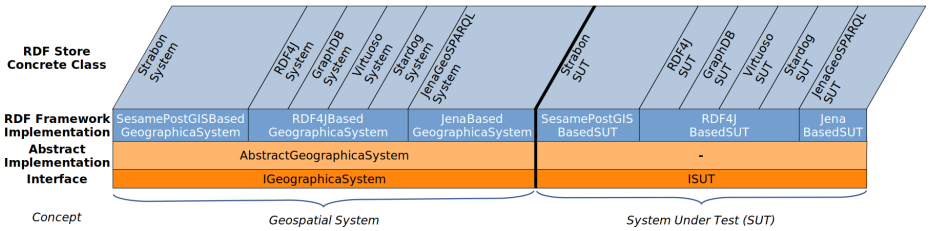

Fig. 4: Systems & SUT Class Hierarchies

RDF Framework Specific System Layer This layer consists of three specializations of the `AbstractGeographicaSystem` class, one for each RDF framework supported by the GeoRDFBench runtime. Each class grounds the generic "connection" to the most appropriate interface or class of the RDF framework it implements. Since each framework has more than one major release, which commonly break backward compatibility, the exact version of each RDF framework supported by the runtime was based on its usage by RDF graph stores. The three specialization classes are:

1. *Sesame API* (`SesamePostGISBasedGeographicaSystem`):
   Most scalable RDF solutions based on Sesame, use v2.6.x since support of the RDBMS Sail was deprecated after that. This Sail allowed graph stores to tap, among other things, into geospatial and other capabilities provided by well known DBMSs' such as PostgreSQL with PostGIS. Therefore, this implementation adds: host, port, database name, user and password, to the system properties map and handles them appropriately. The generic "connection" type is replaced with class:
   `org.openrdf.repository.sail.SailRepositoryConnection`
2. *RDF4J API* (`RDF4JBasedGeographicaSystem`):
   Version 4.x of RDF4J is not supported by all systems, requires Java 11 as the bare minimum, removes initialize methods on Repository, Sail APIs and RepositoryManager and upgrades Lucene libraries from 7.7 to 8.5 affecting disk indexing. Version 3.7.x on the other hand is widely supported by all systems while still offering the required functionality. This implementation focuses on the NativeStore Sail and adds the repository base directory, repository name and indexes used. The generic "connection" type is replaced with interface:
   `org.eclipse.rdf4j.repository.RepositoryConnection`
3. *Jena API* (`JenaBasedGeographicaSystem`):
   Since only Jena GeoSPARQL uses this API, we simply chose the most frequently used Jena version in Maven Central[18], from the latest stable branch, which was 3.17.x. Jena Tuple Database[19] (TDB) was preferred over TDB2 as

[18] https://mvnrepository.com/repos/central
[19] https://jena.apache.org/documentation/tdb/index.html

the persistent storage option as it did not raise as many issues during development. This implementation adds the repository base directory, repository name and injects the transaction functionality in the system initialization and termination code. The generic "connection" type is replaced with interface:
`org.apache.jena.rdf.model.Model`

RDF Framework Specific SUT Layer This layer consists of three base implementations of the `ISUT` interface with the corresponding abstract classes: `SesamePostGISBasedSUT`, `RDF4JBasedSUT` and `JenaBasedSUT`. Each class among other things handles the details of initialization and termination of system, application and database server components of the SUT either as a whole or on a component basis. They also invoke system specific query translations, manage and monitor query execution which takes place in a separate thread, such as, enabling timeout for the executing query and handling customized exceptions thrown by different RDF frameworks during the query evaluation phases.

4.3 Experiment Executor

The executor comprises the concrete `Experiment` and abstract `RunSUTExperiment` classes. The subclasses of `RunSUTExperiment` that RDF modules have to implement are the entry points for all experiment run scripts. The `RunSUTExperiment`, parses the script arguments that describe which JSON specifications (see Figure 1) need to be deserialized into experiment component instances, applies queryset filter if needed, configures the SUT with the above and finally launches the `Experiment` run loop. Two actions, performed at the experiment construction time, are the namespace prefix map merging between the corresponding maps of the system, dataset and queryset along with system dependent queryset rewrites in case non standard vocabularies are used offering similar functionality.

4.4 JSON Specification Generator

This runtime component is a collection of runnable utility classes with no parameters, one for each experiment component type, which create all the specifications necessary to run the Geographica 2 benchmark. This is a geospatial benchmark which the majority of dataset, queryset and execution specifications' types. The user can use them as templates to create new JSON specifications by copying the most similar one and appropriately modifying it. These JSON specifications are part of the project build tree and are readily available to the user.

A more involved researcher can also use the utility class code as examples for easily constructing component specifications for other benchmarks. This is an excellent top-down approach for getting acquainted with the GeoRDFBench's serialization/deserialization capabilities.

Internally, the Jackson[20] JSON library is used to annotate interfaces and class hierarchies to simplify serialization and deserialization. For example, when

[20] https://github.com/FasterXML/jackson

deserializing through a `IGeospatialWorkLoadSpec` workload interface variable, the runtime knows that it should use the concrete `SimpleGeospatialWL` to do so, since we used the **@JsonDeserialize** annotation with the concrete class name as the argument for the `"as"` annotation parameter.

```
@JsonDeserialize(as = SimpleGeospatialWL.class)
public interface IGeospatialWorkLoadSpec {...}
```

The detailed component type hierarchies, create the need to handle many polymorphic instances which must be properly serialized, otherwise it will be impossible to deserialize them. As an example, the serialization process of the class hierarchy starting at the `SimpleGeospatialWL` class is assisted by placing the **@JsonTypeInfo** annotation.

```
@JsonTypeInfo(use = JsonTypeInfo.Id.CLASS,
include = JsonTypeInfo.As.PROPERTY, property = "classname")
public class SimpleGeospatialWL implements IGeospatialWorkLoadSpec {...}
```

The annotation's parameter values specify that the full class name of any polymorphic instance will be also included as an extra JSON property "classname", as it is shown below:

```
{ "classname" : "...runtime.workloadspecs.impl.SimpleGeospatialWL",
"name" : "CensusMacroGeo",
"relativeBaseDir" : "", ...}
```

5 Conclusions and Future Work

We presented the concepts and architecture of the GeoRDFBench Framework, which aims to: (i) save the researcher's time and effort testing new systems, (ii) minimize the margin for errors, (iii) increase reproducibility and results' verification, while (iv) remaining extensible. The 6 implemented RDF Modules provide ample and concrete evidence that introducing a new system requires the absolute necessary user coding to handle only additional properties or deviating system behaviors. Jena GeoSPARQL and GraphDB required the least and most trivial coding. Stardog and Virtuoso required additional code to handle the server aspects of their architecture and query translation to handle non-compliance to the GeoSPARQL standard. Source code, running examples and instructions are provided in our sites[21].

Future work will include support for Hadoop file system and a fourth Spark-based framework API, so that Spark-based distributed GeoSPARQL solutions can be tested.

[21] https://github.com/tioannid/geordfbench, https://geordfbench.di.uoa.gr

References

1. Angles, R., Arenas, M., Barceló, P., Boncz, P.A., Fletcher, G.H.L., Gutierrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J.F., van Rest, O., Voigt, H.: G-CORE: A core for future graph query languages. In: Das, G., Jermaine, C.M., Bernstein, P.A. (eds.) Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018. pp. 1421–1432. ACM (2018). https://doi.org/10.1145/3183713.3190654, https://doi.org/10.1145/3183713.3190654
2. Battle, R., Kolas, D.: Enabling the geospatial Semantic Web with Parliament and GeoSPARQL. Semantic Web **3**(4), 355–370 (2012)
3. Bellini, P., Nesi, P.: Performance assessment of RDF graph databases for smart city services. J. Vis. Lang. Comput. **45**, 24–38 (2018). https://doi.org/10.1016/j.jvlc.2018.03.002, https://doi.org/10.1016/j.jvlc.2018.03.002
4. Bilidas, D., Ioannidis, T., Mamoulis, N., Koubarakis, M.: Strabo 2: Distributed management of massive geospatial rdf datasets. In: The Semantic Web–ISWC 2022: 21st International Semantic Web Conference, Virtual Event, October 23–27, 2022, Proceedings. pp. 411–427. Springer (2022)
5. Bizer, C., Schultz, A.: The berlin sparql benchmark. International Journal on Semantic Web and Information Systems (IJSWIS) **5**(2), 1–24 (2009)
6. Conrads, F., Lehmann, J., Saleem, M., Morsey, M., Ngonga Ngomo, A.C.: I guana: a generic framework for benchmarking the read-write performance of triple stores. In: The Semantic Web–ISWC 2017: 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II 16. pp. 48–65. Springer (2017)
7. Dsouza, A., Tempelmeier, N., Yu, R., Gottschalk, S., Demidova, E.: WorldKG: A world-scale geographic knowledge graph. In: CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021. pp. 4475–4484. ACM (2021)
8. Dunning, A., de Smaele, M., Böhmer, J.: Are the FAIR data principles fair? Int. J. Digit. Curation **12**(2), 177–195 (2017). https://doi.org/10.2218/ijdc.v12i2.567, https://doi.org/10.2218/ijdc.v12i2.567
9. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat, A., Pham, M.D., Boncz, P.: The ldbc social network benchmark: Interactive workload. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 619–630 (2015)
10. Erling, O., Mikhailov, I.: Virtuoso: Rdf support in a native rdbms. In: Semantic web information management: a model-based perspective, pp. 501–519. Springer (2009)
11. Finkel, R.A., Bentley, J.L.: Quad trees a data structure for retrieval on composite keys. Acta informatica **4**, 1–9 (1974)
12. Garbis, G., Kyzirakos, K., Koubarakis, M.: Geographica: A benchmark for geospatial rdf stores (long version). In: The Semantic Web–ISWC 2013: 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II 12. pp. 343–359. Springer (2013)
13. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Journal of Web Semantics **3**(2-3), 158–182 (2005)
14. Hellerstein, J.M., Naughton, J.F., Pfeffer, A.: Generalized search trees for database systems. In: Dayal, U., Gray, P.M.D., Nishio, S. (eds.) VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995,

Zurich, Switzerland. pp. 562–573. Morgan Kaufmann (1995), http://www.vldb.org/conf/1995/P562.PDF
15. Ioannidis, T., Garbis, G., Kyzirakos, K., Bereta, K., Koubarakis, M.: Evaluating geospatial rdf stores using the benchmark geographica 2. Journal on Data Semantics **10**(3-4), 189–228 (2021)
16. Janowicz, K., Hitzler, P., Li, W., Rehberger, D., Schildhauer, M., Zhu, R., Shimizu, C., Fisher, C.K., Cai, L., Mai, G., Zalewski, J., Zhou, L., Stephen, S., Estrecha, S.G., Mecum, B.D., Lopez-Carr, A., Schroeder, A., Smith, D., Wright, D.J., Wang, S., Tian, Y., Liu, Z., Shi, M., D'Onofrio, A., Gu, Z., Currier, K.: Know, know where, knowwheregraph: A densely connected, cross-domain knowledge graph and geo-enrichment service stack for applications in environmental intelligence. AI Mag. **43**(1), 30–39 (2022). https://doi.org/10.1609/aimag.v43i1.19120, https://doi.org/10.1609/aimag.v43i1.19120
17. Jiménez-Ruiz, E., Saveta, T., Zamazal, O., Hertling, S., Roder, M., Fundulaki, I., Ngomo, A.N., Sherif, M., Annane, A., Bellahsene, Z., et al.: Introducing the hobbit platform into the ontology alignment evaluation campaign. In: 13th International Workshop on Ontology Matching (OM). vol. 2288, pp. 49–60 (2018)
18. Jovanovik, M., Homburg, T., Spasić, M.: A geosparql compliance benchmark. ISPRS International Journal of Geo-Information **10**(7), 487 (2021)
19. Karalis, N., Mandilaras, G.M., Koubarakis, M.: Extending the YAGO2 knowledge graph with precise geospatial knowledge. In: Ghidini, C., Hartig, O., Maleshkova, M., Svátek, V., Cruz, I.F., Hogan, A., Song, J., Lefrançois, M., Gandon, F. (eds.) The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11779, pp. 181–197. Springer (2019). https://doi.org/10.1007/978-3-030-30796-7_12, https://doi.org/10.1007/978-3-030-30796-7_12
20. Kostopoulos, C., Mouchakis, G., Troumpoukis, A., Prokopaki-Kostopoulou, N., Charalambidis, A., Konstantopoulos, S.: Kobe: Cloud-native open benchmarking engine for federated query processors. In: The Semantic Web: 18th International Conference, ESWC 2021, Virtual Event, June 6–10, 2021, Proceedings 18. pp. 664–679. Springer (2021)
21. Kyzirakos, K., Karpathiotakis, M., Koubarakis, M.: Strabon: A semantic geospatial dbms. In: The Semantic Web–ISWC 2012: 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I 11. pp. 295–311. Springer Berlin Heidelberg (2012)
22. Manolis Koubarakis (ed.): Geospatial data science: a hands-on approach based on geospatial technologies. ACM Books (2023)
23. Matthew Perry, John Herring: OGC GeoSPARQL - A Geographic Query Language for RDF Data. OGC Implementation Standard OGC 11-052r4, Open Geospatial Consortium (Sep 2012), http://www.opengis.net/doc/IS/geosparql/1.0
24. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.C.: Dbpedia sparql benchmark–performance assessment with real queries on real data. In: The Semantic Web–ISWC 2011: 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I 10. pp. 454–469. Springer (2011)
25. Ngomo, A.C.N., Auer, S., Lehmann, J., Zaveri, A.: Introduction to linked data and its lifecycle on the web. Reasoning Web. Reasoning on the Web in the Big Data Era: 10th International Summer School 2014, Athens, Greece, September 8-13, 2014. Proceedings 10 pp. 1–99 (2014)
26. Osman, T., Albiston, G.: Geosparql-jena: Implementation and benchmarking of a geosparql graphstore. In: 23rd European Conference on Knowledge Management Vol 2. Academic Conferences and publishing limited (2022)

27. Patroumpas, K., Giannopoulos, G., Athanasiou, S.: Towards geospatial semantic data management: strengths, weaknesses, and challenges ahead. In: Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. pp. 301–310 (2014)
28. Perry, M., Estrada, A., Das, S., Banerjee, J.: Developing geosparql applications with oracle spatial and graph. In: SSN-TC/OrdRing@ ISWC. pp. 57–61 (2015)
29. van Rest, O., Hong, S., Kim, J., Meng, X., Chafi, H.: PGQL: a property graph query language. In: Boncz, P.A., Larriba-Pey, J.L. (eds.) Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016. p. 7. ACM (2016). https://doi.org/10.1145/2960414.2960421, https://doi.org/10.1145/2960414.2960421
30. Röder, M., Kuchelev, D., Ngomo, A.N.: HOBBIT: A platform for benchmarking big linked data. Data Sci. **3**(1), 15–35 (2020). https://doi.org/10.3233/ds-190021, https://doi.org/10.3233/ds-190021
31. Rodriguez, M.A.: The gremlin graph traversal machine and language (invited talk). In: Proceedings of the 15th Symposium on Database Programming Languages. ACM (oct 2015). https://doi.org/10.1145/2815072.2815073, https://doi.org/10.1145%2F2815072.2815073
32. Saleem, M., Mehmood, Q., Ngonga Ngomo, A.C.: Feasible: A feature-based sparql benchmark generation framework. In: The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I 14. pp. 52–69. Springer (2015)
33. Thakkar, H.: Towards an open extensible framework for empirical benchmarking of data management solutions: Litmus. In: The Semantic Web: 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28–June 1, 2017, Proceedings, Part II 14. pp. 256–266. Springer (2017)
34. Troumpoukis, A., Konstantopoulos, S., Mouchakis, G., Prokopaki-Kostopoulou, N., Paris, C., Bruzzone, L., Pantazi, D.A., Koubarakis, M.: Geofedbench: A benchmark for federated geosparql query processors. In: ISWC (Demos/Industry). pp. 228–232 (2020)
35. Wilkinson, M.D., Dumontier, M., Aalbersberg, I.J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.W., da Silva Santos, L.B., Bourne, P.E., et al.: The fair guiding principles for scientific data management and stewardship. Scientific data **3**(1), 1–9 (2016)