The GeoRDFBench Framework: Geospatial semantic benchmarking simplified

1st Theofilos Ioannidis
Department of Informatics and Telecommunications
National and Kapodistrian University of Athens
Athens, Greece
tioannid@di.uoa.gr

2nd Manolis Koubarakis
Department of Informatics and Telecommunications
National and Kapodistrian University of Athens
Athens, Greece
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.**

***Index Terms*—geospatial, semantic, benchmarking, framework**

I. Introduction

Many projects [1]–[3] have shown 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 [4]. Graph database systems with varying degrees of spatial support have recently been used to manage very large LOD datasets [5]–[13]. Selecting the most suitable system for user 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 [14]–[18] have assisted in creating datasets of desired size and attributes, while log-mining techniques [16], [18], [19] 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 [7], [10], [17] 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[1] 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

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

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 features a runtime, which *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*. The runtime contains the *JSON generator whose API enables the creation of serialized versions of all benchmark component specifications*. The framework comes pre-bundled with *a library of JSON specifications which include the Geographica 2 benchmark's components, a PostgreSQL implementation of the JDBC report sink and hosts with Linux operating system*.
2) The runtime *abstracts and generalizes repository connection functionality, from the three best known RDF framework APIs*: (i) OpenRDF Sesame, (ii) Eclipse RDF4J and (iii) Apache Jena. At the same time, *it explicitly models the RDF store application and back-end modules by embedding their life-cycle management in the experiment execution*. It combines the above by *implementing class hierarchies of geospatial enabled RDF stores* according to their basic module architecture and the RDF framework APIs they support.
3) The framework *provides example implementations for geospatial RDF stores* of different architectures and different RDF framework APIs. These come in the form of Maven modules[2] and the list includes: *(i) RDF4J with Lucene SAIL, (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 effort is trivial* as most of the heavy lifting has been pulled upwards in the class hierarchies of the runtime.
4) The runtime experiment executor features a *single experiment execution loop* independent of the store architecture and RDF framework API used. It enables *automatic result accuracy verification* for workloads that have embedded an expected resultset, while query execution accuracy results are persisted with other statistics. The executor features *programmable timeout* and a *fine grained error handling* mechanism during each one of the query execution phases, which allows it to *measure results separately from scan errors* and *minimize the execution abort scenarios*. It features *query result sampling in experiment logs* for verification and debugging purposes. It allows *statistics customization* and *synchronous or deferred experiment results' persistence* to a user-defined report sink.
5) The framework can also be used for *SPARQL benchmarks*. LUBM [14] benchmark's workload component is included in the JSON specification library.

[2]GeoRDFBench framework is a Java Maven multi-module POM project.

The organization of the rest of the paper is as follows. Section II discusses related work. Section III presents the high level architecture of the framework while section IV explains in more detail the GeoRDFBench's runtime. Showcasing the JSON generator API follows in section V. Section VI shows an evaluation of the framework. Finally, section VII presents conclusions and future work.

II. Related Work and Background

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 Resource Description Framework (RDF) or the Labeled Property Graph (LPG) approach. RDF as a data model has good expressivity, while featuring a standardized declarative query language SPARQL [20] and a standardized spatial vocabulary GeoSPARQL [21]. LPGs, on the other hand, excel in graph traversal and path search for analytics and machine learning. But, until very recently, they lacked standardization as there are several data models and languages from high-profile vendors and institutions, such as, Neo4j's Cypher [22], Apache TinkerPop Gremlin [23], the Oracle supported PGQL [24] and G-Core [25] from the Linked Data Benchmark Council (LDBC). Another issue is that some of these languages are declarative while others are procedural. As of April 2024, the first edition of the Graph Query Language (GQL) standard [26] is officially published and hopefully will allow language inter-operation with SPARQL. To conclude, at the time of writing of this paper, most geospatial graph databases that support complex geometries support the RDF model.

Geospatial Graph Store Architecture: The available geospatial graph stores feature a 3-tier architecture with standard and optional components.

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 [8] uses the embedded key-value database Berkeley DB with a standard R-tree spatial index, Strabon [6] uses PostgreSQL and PostGIS with an R-tree over GiST [27] 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 [28] 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 [29] and geohash [30] 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 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 real-world 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[3], we proceed with large real-world 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 [14], BSBM [15], DBpedia SPARQL benchmark (DBPSB) [16] and the Social Network Benchmark (SNB) [18], just to mention a few. GeoSPARQL benchmarks have been presented in [31], [32], the benchmark Geographica in [17], a smart city services related benchmark in [33], a compliance benchmark in [34] and Geographica 2 in [10].

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

[3]Removing problematic queries or reconfiguring the system.

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 query-set and dataset generation task. For example, 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 [19] 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 [35] 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 [36] 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 [37] cloud benchmarking engine for federated query processors includes the GeoFedBench [38] benchmark, which focuses on validation of the actual crop land usage against the Austrian land survey dataset.

Benchmarking Framework Platforms: A more recent idea is that of *benchmarking framework platforms* which are 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 [39], the most complete of these platforms, extends the scope of benchmarking to the entire linked data life-cycle [40], such as *link discovery* [41], employs intuitive web UIs and allows the integration of systems in various programming languages. Overall, 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 *"FAIR Data Principles in Benchmarking"* opinions section of the GeoRDFBench Framework site [42].

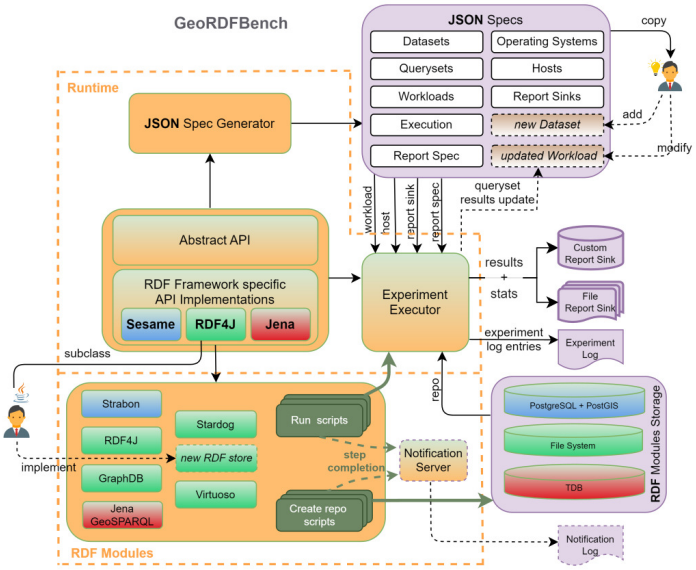

Fig. 1: Architectural overview of GeoRDFBench Framework

III. 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.

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 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*

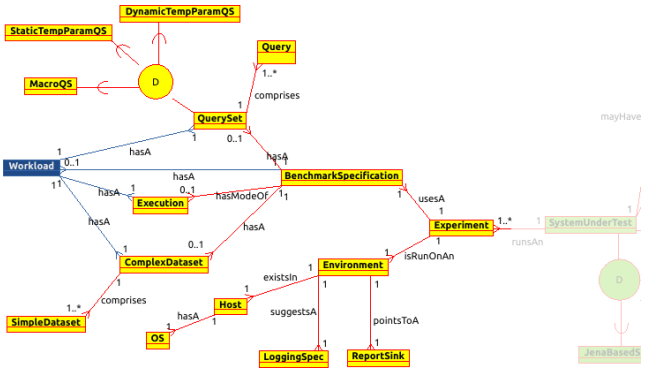

Fig. 2: EER diagram of Abstract API - Experiment Components

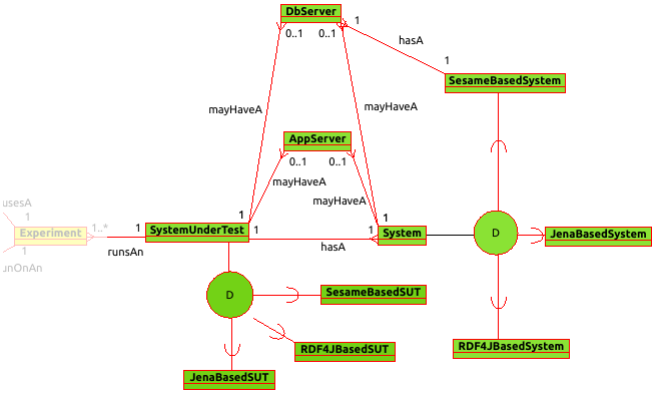

Fig. 3: EER diagram of Abstract API - System Under Test

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 and LUBM benchmarks' components and are generated by the *JSON Specs Generator*. This "starter dough" library is stored on the file system separately from GeoRDFBench code and can be easily copied or modified by the user.

IV. GeoRDFBench Runtime: The Framework's 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.

A. Abstract API

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[4] 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.

1) Experiment Components: Figure 2 depicts the experiment components, which are detailed below and which are logically organized into the *Benchmark Specification* and *Experiment Environment* groups.

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*.

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

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

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.

***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"[5] experiment types, the `MacroQS` subclass and its specializations[6] are useful for "macro"[7] experiment types.

***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.

***Workload**(compact form):* The entity `Workload`, in Figure 2, represents the compact form of the benchmark experiment description: *dataset + queryset + execution specification*.

Experiment Environment: This group includes all experiment platform dependent components.

***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).

***Host**:* The entity `Host`, in Figure 2, represents the hardware platform where the benchmark experiment is taking place. It has the host name, the operating system, IP address, total RAM (GBs), the base path for the actual dataset files, the base path for RDF store repository files and the base path for the default reports and statistics.

***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. The target report sink database schema is generated with the help of the runtime-bundled *database generation SQL script*.

Logging specification[8]*:* 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.

2) 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

[5]Independent queries, each run several times with cold or warm caches.

[6]Not shown in Figure 2 for simplicity reasons.

[7]A sequence of queries representing a case scenario, that is run repeatedly as a whole.

[8]Also mentioned as ReportSpec in the framework

the other hand represents the combination of 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.

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. 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. 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.

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.

B. 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: (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.

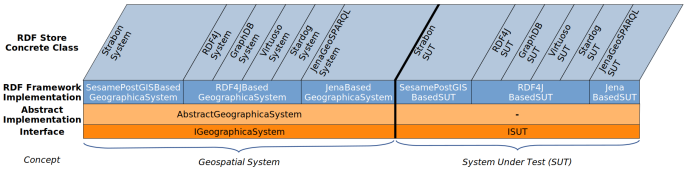

Fig. 4: Systems & SUT Class Hierarchies

1) 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:

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
```

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
```

Jena API *(`JenaBasedGeographicaSystem`):* Since only Jena GeoSPARQL uses this API, we simply chose the most frequently used Jena version in Maven Central [43] , from the latest stable branch, which was 3.17.x. Jena Tuple Database (TDB) [44] was preferred over TDB2 as 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 "con-

nection" type is replaced with interface:

```
org.apache.jena.rdf.model.Model
```

2) 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.

C. 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.

D. 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 geospatial benchmark employs the majority of dataset, queryset and execution specifications' types. These JSON specifications are part of the project build tree and are readily available to the user.

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. For this purpose, the Jackson [45] JSON library is used to annotate interfaces and class hierarchies to simplify serialization and deserialization.

V. JSON Generator API - By Example

In this section, we demonstrate the use of the runtime JSON generator API for generating the *detailed form* for benchmark specifications, using as test case, the *Scalability* workload of the Geographica 2 GeoSPARQL benchmark.

Scalability Workload Description: This workload (see Table I) is intended to evaluate geospatial RDF store scalability against increasingly larger real-world datasets with many complex geometries. It comprises 6 datasets with increasing number of triples (10K, 100K, 1M, 10M, 100M, 500M), a set of 3 spatial function queries (1 selection and 2 joins) and a similar set of spatial predicate queries[9]. The experiment execution model defines that each query shall be executed 3 times with COLD caches, 3 times with WARM caches and that the median of the 3 execution times shall be considered as the result for COLD and WARM executions. The maximum timeout period for each query is set to 24 hours and in case a query's COLD execution fails then all remaining COLD and WARM executions should be skipped. A 5000 msec delay is also specified after clearing caches and waiting for garbage collection. The below code samples demonstrate how to generate three different types of specifications, serialize them to JSON files and then deserialize them from the same files.

TABLE I: Geographica 2 Scalability workloads

Workload	Dataset	Queryset	Execution Spec
Scalability 10K	***scalability_10K***	***scalabilityFunc*** or scalabilityPred	***scalability***
Scalability 100K	scalability_100K		
Scalability 1M	scalability_1M		
Scalability 10M	scalability_10M		
Scalability 100M	scalability_100M		
Scalability 500M	scalability_500M		

A. Dataset generation

The following code creates the `scalability_10K` complex dataset specification object.

Listing 1: Generate Dataset Specification

```
public static GeographicaDS newScalabilityDS() {
  // create a simple dataset object with a single N-Triples file
  GenericGeospatialSimpleDS sds
    = new GenericGeospatialSimpleDS("scalability_10K", // simple dataset name
       "Scalability/10K", // relative directory where the file resizes
       "scalability_10K.nt", NTRIPLES_STR); // the triples file
  // add to it any namespace prefixes used in the dataset file
  sds.addUsefulNamespacePrefix("lgo", "<http://data.linkedeodata.eu/ontology#>");
  // add to it any property used in the dataset that denotes feature geometry
  sds.addHasGeometry("scalabilityHasGeometry",
         "<http://www.opengis.net/ont/geosparql#hasGeometry>");
  // add to it any property used in the dataset that denotes WKT serialization
  sds.addAsWKT("scalabilityAsWKT", "<http://www.opengis.net/ont/geosparql#asWKT>");
  // create a complex dataset object with a single simple dataset object
  GeographicaDS gds =
    GeographicaDS(sds, // the simple dataset
       "", // context/graph IRI for the simple dataset
       0); // synthetic dataset scaling factor, 0 for non synthetic datasets
  // serialize the complex dataset specification object to a JSON file
  gds.serializeToJSON(new File(SCALABILITY_JSONDEF_FILE));
  // deserialize a complex dataset objet from a JSON file and return it
  return DataSetUtil.deserializeFromJSON(SCALABILITY_JSONDEF_FILE);
}
```

[9]Systems, such as GraphDB and Parliament, use their spatial index only with spatial predicates.

The complex dataset comprises an N-Triples simple dataset file.

Listing 2: Serialized Dataset Specification

```
{ "classname" : "gr.uoa.di.rdf.Geographica3.runtime.datasets.complex.impl.GeographicaDS",
"name" : "scalability_10K",
"relativeBaseDir" : "Scalability/10K",
"simpleGeospatialDataSetList" : [ {
  "name" : "scalability_10K",
  "relativeBaseDir" : "Scalability/10K",
  "dataFile" : "scalability_10K.nt",
  "rdfFormat" : "N-TRIPLES",
  "mapUsefulNamespacePrefixes" : {
    "geo" : "<http://www.opengis.net/ont/geosparql#>",
    "rdf" : "<http://www.w3.org/1999/02/22-rdf-syntax-ns#>",
    "owl" : "<http://www.w3.org/2002/07/owl#>",
    "geof" : "<http://www.opengis.net/def/function/geosparql/>",
    "lgo" : "<http://data.linkedeodata.eu/ontology#>",
    "xsd" : "<http://www.w3.org/2001/XMLSchema#>",
    "rdfs" : "<http://www.w3.org/2000/01/rdf-schema#>",
    "geo-sf" : "<http://www.opengis.net/ont/sf#>"
  },
  "mapAsWKT" : {
    "scalabilityAsWKT" : "<http://www.opengis.net/ont/geosparql#asWKT>"
  },
  "mapHasGeometry" : {
    "scalabilityHasGeometry" : "<http://www.opengis.net/ont/geosparql#hasGeometry>"
  }
} ],
"mapDataSetContexts" : {
  "scalability_10K" : ""
},
"n" : 0}
```

B. Queryset generation

We also generate the first variant of the queryset, `scalabilityFunc`, which uses spatial functions.

Listing 3: Generate Queryset Specification

```
public static StaticTempParamQS newScalabilityFuncQS() {
  // read fixed Polygon from external file which is used in spatial selection query
  String givenPolygon = readFile(SCALABILITY_EUROPE_POLYGON_FILE);
  // initialize the map of useful general RDF related prefixes
  Map<String, String> mapUsefulNamespacePrefixes = new HashMap<>();
  // initialize the map of template parameters
  Map<String, String> mapTemplateParams = new HashMap<>();
  mapTemplateParams.put("FUNCTION", "sfIntersects");
  mapTemplateParams.put("GIVEN_SPATIAL_LITERAL", givenPolygon);
  // populate Graph prefixes map
  Map<String, String> mapLiteralValues = new HashMap<>();
  // populate template queries map
  Map<Integer, IQuery> mapQry = new HashMap<>();
  mapQry.put(0, new SimpleQuery("SC1_Geometries_Intersects_GivenPolygon",
"SELECT ?s1 ?o1 WHERE { \n ?s1 geo:asWKT ?o1 . \n FILTER(geof:FUNCTION(?o1,
       GIVEN_SPATIAL_LITERAL)). \n} \n",
false));
  mapQry.put(1, new SimpleQuery("SC2_Intensive_Geometries_Intersect_Geometries",
"SELECT ?s1 ?s2 \nWHERE { \n ?s1 geo:hasGeometry [ geo:asWKT ?o1 ] ;\n lgo:has_code
       \"1001\"^^xsd:integer . \n ?s2 geo:hasGeometry [ geo:asWKT ?o2 ] ;\n lgo:has_code
       ?code2 . \n FILTER(?code2>5000 && ?code2<6000 && ?code2 != 5260) .\n
       FILTER(geof:FUNCTION(?o1, ?o2)). \n} \n",
false));
  mapQry.put(2, new SimpleQuery("SC3_Relaxed_Geometries_Intersect_Geometries",
"SELECT ?s1 ?s2\nWHERE {\n ?s1 geo:hasGeometry [ geo:asWKT ?o1 ] ;\n lgo:has_code
       \"1001\"^^xsd:integer .\n ?s2 geo:hasGeometry [ geo:asWKT ?o2 ] ;\n lgo:has_code
       ?code2 .\n FILTER(?code2 IN (5622, 5601, 5641, 5621, 5661)) .\n
       FILTER(geof:FUNCTION(?o1, ?o2)).\n} \n",
false));
  StaticTempParamQS scalabilityQS =
    new StaticTempParamQS("scalabilityFunc", "", false,  mapQry,
      mapTemplateParams, mapUsefulNamespacePrefixes, mapLiteralValues);
  // serialize the queryset specification object to a JSON file
  scalabilityQS.serializeToJSON(new File(SCALABILITY_FUNC_JSONDEF_FILE));
  // deserialize a queryset objet from a JSON file and return it
  return QuerySetUtil.deserializeFromJSON(SCALABILITY_FUNC_JSONDEF_FILE);
}
```

We are using the `StaticTempParamQS` subclass which allows the user to model querysets with or without template parameters but with fixed parameter values for all queries.

Listing 4: Serialized Queryset Specification

```
{ "classname" :
       "gr.uoa.di.rdf.Geographica3.runtime.querysets.complex.impl.StaticTempParamQS",
"name" : "scalabilityFunc",
"relativeBaseDir" : "",
"hasPredicateQueriesAlso" : false,
"mapQueries" : {
  "0" : {
    "label" : "SC1_Geometries_Intersects_GivenPolygon",
    "text" : "SELECT ?s1 ?o1 WHERE { \n ?s1 geo:asWKT ?o1 . \n FILTER(geof:FUNCTION(?o1,
           GIVEN_SPATIAL_LITERAL)). \n} \n",
    "usePredicate" : false,
    "expectedResults" : -1
},
  "1" : {
    "label" : "SC2_Intensive_Geometries_Intersect_Geometries",
    "text" : "SELECT ?s1 ?s2 \nWHERE { \n ?s1 geo:hasGeometry [ geo:asWKT ?o1 ] ;\n
           lgo:has_code \"1001\"^^xsd:integer . \n ?s2 geo:hasGeometry [ geo:asWKT ?o2 ]
           ;\n lgo:has_code ?code2 . \n FILTER(?code2>5000 && ?code2<6000 && ?code2 !=
           5260) .\n FILTER(geof:FUNCTION(?o1, ?o2)). \n} \n",
    "usePredicate" : false,
    "expectedResults" : -1
    },
  "2" : {
    "label" : "SC3_Relaxed_Geometries_Intersect_Geometries",
    "text" : "SELECT ?s1 ?s2\nWHERE {\n ?s1 geo:hasGeometry [ geo:asWKT ?o1 ] ;\n
           lgo:has_code \"1001\"^^xsd:integer .\n ?s2 geo:hasGeometry [ geo:asWKT ?o2 ] ;\n
           lgo:has_code ?code2 .\n FILTER(?code2 IN (5622, 5601, 5641, 5621, 5661)) .\n
           FILTER(geof:FUNCTION(?o1, ?o2)).\n} \n",
    "usePredicate" : false,
    "expectedResults" : -1
  }
},
"mapUsefulNamespacePrefixes" : { },
"mapTemplateParams" : {
  "GIVEN_SPATIAL_LITERAL" : "\"POLYGON((23.708496093749996
         37.95719224376526,22.906494140625 40.659805938378526,11.524658203125002
         48.16425348854739,-0.1181030273437499 51.49506473014367,-3.2189941406250004
         55.92766341247031,-5.940856933593749 54.59116279530599,-3.1668090820312504
         51.47967237816337,23.708496093749996
         37.95719224376526))\"^^<http://www.opengis.net/ont/geosparql#wktLiteral>",
  "FUNCTION" : "sfIntersects"
},
"mapGraphPrefixes" : { } }
```

C. Execution spec generation

The following code generates the Scalability execution specification object.

Listing 5: Generate Execution Specification

```
public static SimpleES newScalabilityES() {
  // create a map with no of executions per execution type
  Map<ExecutionType, Integer> execTypeReps = new HashMap<>();
  execTypeReps.put(ExecutionType.COLD, 3);
  execTypeReps.put(ExecutionType.WARM, 3);
  // create a simple execution specification object
  SimpleES sses = new SimpleES(execTypeReps,
    24 * 60 * 60, // 24 hours max duration per query execution
    7 * 24 * 60 * 60, // 7 days max duration for the experiment
    Action.RUN, // run experiments instead or printing ground queryset
    AverageFunction.QUERY_MEDIAN, // use median instead of mean
    BehaviourOnColdExecutionFailure.SKIP_REMAINING_ALL_QUERY_EXECUTIONS,
    5000); // 5000 msecs delay for clearing caches and garbage collection
  // serialize the execution specification object to a JSON file
  sses.serializeToJSON(new File(SCALABILITYEXECUTIONSPECJSONDEF_FILE));
  // deserialize an execution spec object from a JSON file and return it
  return ExecutionSpecUtil.deserializeFromJSON(SCALABILITYEXECUTIONSPECJSONDEF_FILE);
}
```

The below serialized representation fully describes the experiment execution model in *Scalability Workload Description* presented earlier in this section.

Listing 6: Serialized Execution Specification

```
{ "classname" : "gr.uoa.di.rdf.Geographica3.runtime.executionspecs.impl.SimpleES",
"execTypeReps" : {
  "COLD" : 3,
  "WARM" : 3
},
"maxDurationSecsPerQueryRep" : 86400,
"maxDurationSecs" : 604800,
"action" : "RUN",
"avgFunc" : "QUERY_MEDIAN",
"onColdFailure" : "SKIP_REMAINING_ALL_QUERY_EXECUTIONS",
"clearCacheDelaymSecs" : 5000}
```

In a similar manner, the user can generate the experiment environment specifications: host, operating system, report sink and logging. At this point we should remind that, as it was envisioned by design, the trivial

method of copying and modifying an existing specification file with a standard text editor, will probably suffice for many use cases, once a library of JSON specifications is already available.

For an example of generating the *compact form* of a benchmark specification, the user can refer to *GeoRDFBench Framework Samples* [46] application that is using as test case, the LUBM benchmark for SPARQL.

VI. Experimental Evaluation

In this section, we present the process of running some of the Geographica 2 [10] benchmark scalability experiments with the help of the GeoRDFBench framework and present the results. The experiment environment, benchmark description and execution details are presented below.

A. Environment

The hardware platform for the experiment was an Intel NUC8i7BEH box, Ubuntu 22.04.2 LTS with 32GB DDR4-2400MHz, a Samsung SSD NVMe 970 EVO Plus 500GB system disk and a secondary data disk Western Digital WDC WD20SPZX-75U 2TB mounted on `/data`. Both filesystems `/` and `/data` were formatted as "ext4". All SUTs and their repository data were intentionally placed under the slower `/data` filesystem.

In addition PostgreSQL v14.7 with PostGIS v3 was also installed, since Strabon requires it for creating spatial databases. The GeoRDFBench was uncompressed in `/data/GeoRDFBench`. The project come ready with prebuilt binaries. To setup the report sink, we run the `scripts/geographica3.sql` that creates the corresponding database:

```
user@NUC8i7BEH:$/data/GeoRDFBench$ sudo -i -u postgres
postgres@NUC8i7BEH:~$ psql -g /data/GeoRDFBench/scripts/geographica3.sql
```

Choice of Systems: We chose the following 4 systems to participate in this demonstration: (i) GraphDB, (ii) RDF4J, (iii) Stardog and (iv) Strabon. The first three use the RDF4J Framework while the last one uses OpenRDF Sesame. Stardog and GraphDB also required that we install the corresponding server software and their licenses under `/data`.

B. Benchmark description

We chose the 10K, 100K and 1M variants of the Scalability workload (see Table I). The following listing shows the ground queryset for RDF4J:

Listing 7: RDF4J Ground Queries & Prefix Header

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geo-sf: <http://www.opengis.net/ont/sf#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX lgo: <http://data.linkedeodata.eu/ontology#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

# Query 0 - SC1_Geometries_Intersects_GivenPolygon:
SELECT ?s1 ?o1 WHERE {
  ?s1 geo:asWKT ?o1 .
  FILTER(geof:sfIntersects(?o1, "POLYGON((23.708496093749996 37.95719224376526,
    ... 37.95719224376526))"^^<http://www.opengis.net/ont/geosparql#wktLiteral>)).
}
# Query 1 - SC2_Intensive_Geometries_Intersect_Geometries:
SELECT ?s1 ?s2 WHERE {
  ?s1 geo:hasGeometry [ geo:asWKT ?o1 ] ;
    lgo:has_code "1001"^^xsd:integer .
  ?s2 geo:hasGeometry [ geo:asWKT ?o2 ] ;
    lgo:has_code ?code2 .
  FILTER(?code2>5000 && ?code2<6000 && ?code2 != 5260) .
  FILTER(geof:sfIntersects(?o1, ?o2)).
}
# Query 2 - SC3_Relaxed_Geometries_Intersect_Geometries:
SELECT ?s1 ?s2 WHERE {
  ?s1 geo:hasGeometry [ geo:asWKT ?o1 ] ;
    lgo:has_code "1001"^^xsd:integer .
  ?s2 geo:hasGeometry [ geo:asWKT ?o2 ] ;
    lgo:has_code ?code2 .
  FILTER(?code2 IN (5622, 5601, 5641, 5621, 5661)) .
  FILTER(geof:sfIntersects(?o1, ?o2)).
}
```

GeoRDFBench, during experiment execution, *automatically constructs the prefix header to include system, queryset and dataset specific namespace prefixes*. For example, while Strabon headers do not differ from the one presented above, the headers for GraphDB and Stardog also include the following system specific namespace prefixes:

Listing 8: GraphDB & Stardog Prefix Header Differences

```
# GraphDB adds a few useful extensions based on the USeekM library
PREFIX ext: <http://rdf.useekm.com/ext#>
# Stardog also uses the QUDT ontology for distance units
PREFIX unit: <http://qudt.org/vocab/unit#>
```

To produce the ground queries of Listing 7 GeoRDFBench conveniently takes the necessary steps. First, the template parameters of the queries in Listing 4 are *replaced with their values during queryset deserialization*. After that, it *applies any system provided query translations in order to create the ground queries*. For example, Stardog emulates the GeoSPARQL `geof:sfIntersects` function with the combination of the non-standard `geof:relate` through the negation of `geo:disjoint` as exhibited below for query SC1:

Listing 9: Stardog Queries SC1 Differences

```
# Query 0 - SC1_Geometries_Intersects_GivenPolygon:
SELECT ?s1 ?o1 WHERE {
  ?s1 geo:asWKT ?o1 .
  ?relation geof:relate(?o1 "POLYGON((23.708496093749996 37.95719224376526,
    ... 37.95719224376526))"^^<http://www.opengis.net/ont/geosparql#wktLiteral>) .
  FILTER(?relation != geo:disjoint) .
}
```

Such system dependent logic, which may include different vocabularies, has been identified and implemented in the corresponding `translateQuery()` function of the relevant RDF Module SUTs for the user's convenience and provide a more consistent and trouble free experience. StardogSUT appropriately translates the unsupported GeoSPARQL `geof:sfWithin` and `geof:sfEquals` functions. GraphDBSUT handles the non-standard `geof:sfEquals` function behaviour. VirtuosoSUT maps `geof:sfWithin`, `geof:buffer`, `geof:distance`, `geof:sfEquals` functions to `bif:st_within`,

`bif:st_point, bif:st_distance, bif:st_within`, the `geo:wktLiteral` data type to `virtrdf:Geometry` and handles default distance unit differences between functions.

The execution specification used for the scalability queryset is the same as in Listing 6.

C. Repository Creation

We proceeded with the creation of repositories `scalability_{10K,100K,1M}` for the four SUTs. In a separate terminal for each system we executed the following steps:

1) Prepare environment variables: We sourced the **environment preparation script** to prepare the environment variables for running the RDF4JSUT on the NUC8I7BEH host:

```
user@NUC8i7BEH:/data/GeoRDFBench/scripts$ source ./prepareRunEnvironment.sh
 NUC8I7BEH RDF4JSUT "CreateRDF4JRepos"
```

2) Review environment variables: The environment variables can be reviewed with the **print environment script**:

```
user@NUC8i7BEH:/data/GeoRDFBench/scripts$ ./printRunEnvironment.sh
All SUTs
--------
Environment = NUC8I7BEH
DatasetBaseDir = /data/Geographica2_Datasets
ActiveSUT = RDF4JSUT
...
RDF4J SUT
---------
RDF4JRepoBaseDir = /data/RDF4J_3.7.7_Repos/server
Version = 3.7.7
...
```

3) Enable selected repositories: The **repository generation wrapper scripts** for all RDF modules have largely the same functionality. First, they require that the user either provides all the required arguments for the system at hand, or that the environment variables have already been prepared as shown before. Each of these scripts comes, already setup, with a list of repositories to create automatically. The user, most frequently, will want to comment out the repositories that are not required or add new repositories. We enabled the desired repositories for RDF4JSUT by applying the following modifications to `../RDF4JSUT/scripts/CreateRepos/createAllRDF4JRepos.sh`:

Listing 10: Modifying RDF4J RepoGen Wrapper Script

```
RDF4JSUT/scripts/CreateRepos$ diff -U 1 createAllRDF4JRepos.sh createAllRDF4JRepos_mod.sh
...
@@ -119,3 +119,3 @@
WKTIdxList="http://www.opengis.net/ont/geosparql#asWKT"
- levels=( "10K" )
+ levels=( "10K" "100K" "1M" )
#levels=( "10K" "100K" "1M" "10M" "100M" "500M" )
@@ -140,2 +140,2 @@
```

4) Generate enabled repositories: The following command created the `scalability_{10K,100K,1M}` RDF4J repositories:

```
user@NUC8i7BEH:/data/GeoRDFBench/RDF4JSUT/scripts/CreateRepos$
./createAllRDF4JRepos.sh false 2>&1 | tee -a createRDF4JRepos.log
```

The generated log details all actions taken on each source dataset file, repository creation durations and repository final sizes. The **false** parameter denotes that we do not wish to overwrite existing repositories.

D. Run expirements

We proceeded with the execution of the workloads for the repositories `scalability_{10K,100K,1M}` for the four SUTs. For each system in a separate terminal we run once the **experiment run script** for the detailed representation. The following annotated command runs the `Scalability 10K` workload for RDF4JSUT on the NUC8i7BEH host.

```
user@NUC8i7BEH:$ export JSONLIB=/data/GeoRDFBench/json_defs
user@NUC8i7BEH:$/data/GeoRDFBench/RDF4JSUT/scripts/RunTests3/runTestsForRDF4JSUT.sh
-Xmx24g  # max RAM allocated
-rbd RDF4J_3.7.7_Repos/server # relative base path for RDF4J repositories
-expdesc 172#_2024-02-20_RDF4JSUT_Run_Scal10K # experiment description
-ds ${JSONLIB}/datasets/scalability_10Koriginal.json # dataset
-qs ${JSONLIB}/querysets/scalabilityFuncQSoriginal.json  # queryset
-es ${JSONLIB}/executionspecs/scalabilityESoriginal.json # execution
-h ${JSONLIB}/hosts/nuc8i7behHOSToriginal.json # host
-rs ${JSONLIB}/reportspecs/simplereportspec_original.json # logging
-rpsr ${JSONLIB}/reportsources/ubuntu_vma_tioaRepSrcoriginal.json # report sink
```

Width sanitized sample output is shown below:

```
...
812[main] INFO  RDF4JSUT  - Initializing..
1157 [main] INFO  Experiment  - RDF4JBasedGeographicaSystem-dependent
translation of the queryset scalability
1157 [main] INFO  Experiment  - RDF4JBasedGeographicaSystem-dependent
namespace prefixes merged with the prefixes of queryset scalability
...
Executing query [0, SC1_Geometries_Intersects_GivenPolygon] (COLD, 0):
SELECT ?s1 ?o1 WHERE {
?s1 geo:asWKT ?o1 .
FILTER(geof:sfIntersects(?o1, "POLYGON((23.708496093749996 37.95719224376526,
...
 37.95719224376526))"^^<http://www.opengis.net/ont/geosparql#wktLiteral>)). }

6333 [main] INFO  GenericLinuxOS  - Clearing caches...
11375 [main] INFO  GenericLinuxOS  - Caches cleared after delay of 5000 msecs
11570 [main] INFO  RDF4JSUT  - Starting QueryExecutor thread
11573 [main] INFO  RDF4JSUT  - Timeout progress step is 21600000 msecs
11573 [Thread-4] INFO  QueryRepResult  - Transitioning (NOTSTARTED => STARTED)
11573 [Thread-4] INFO  QueryRepResult  - Transitioning (STARTED => PREPARING)
11646 [Thread-4] INFO  QueryRepResult  - Transitioning (PREPARING => EVALUATING)
11702 [Thread-4] INFO  QueryRepResult  - Transitioning (EVALUATING => EVALUATED)
11702 [Thread-4] INFO  RDF4JbasedExecutor  -  s1 o1
11702 [Thread-4] INFO  RDF4JbasedExecutor  - ----------------------------------->
11702 [Thread-4] INFO  QueryRepResult  - Transitioning (EVALUATED => SCANNING)
11878 [Thread-4] INFO  - http://data.linkedeodata.eu/osm/wales/transport/Geometry/1686299
"MULTIPOLYGON (((-4.0803447 53.0841743, -4.0803 53.0841838, -4.0802 53.0841, -4.08030
53.08411, -4.08034 53.0841743)))"^^<http://www.opengis.net/ont/geosparql#wktLiteral>
...
12447 [Thread-4] INFO  RDF4JbasedExecutor  - <-----------------------------------
12447 [Thread-4] INFO  QueryRepResult  - Transitioning (SCANNED => COMPLETED)
12447 [main] INFO  RDF4JSUT  - Percentage of expired timeout is 0.0 %
12488 [main] INFO  Experiment  - |<==
Executed query [0, SC1_Geometries_Intersects_GivenPolygon] (COLD, 0):
<COMPLETED-NONE> 56349004 + 744380367 = 800729371 nsecs,
554 results, 0 scan errors - ACCURACY NOT DETERMINED
12488 [main] INFO  RDF4JSUT  - Closing..
12488 [main] INFO  RDF4JBasedGeographicaSystem  - Closing connection...
12489 [main] INFO  RDF4JBasedGeographicaSystem  - Repository closed.
....
143618 [main] INFO  JDBCRepSrc  - Deferred mode for JDBCRepSrc was enabled.
18 records were flushed
143618 [main] INFOGen  - Export statistics in
"../172#_2024-02-20_RDF4JSUT_Run_Scal10K/Scalability/10K/RDF4JSUT-Experiment"
143675 [main] INFOGen  - Statistiscs printed:
../172#_2024-02-20_RDF4JSUT_Run_Scal10K/Scalability
/10K/RDF4JSUT-Experiment/00-SC1_Geometries_Intersects_GivenPolygon-cold
...
../172#_2024-02-20_RDF4JSUT_Run_Scal10K/Scalability
/10K/RDF4JSUT-Experiment/02-SC3_Relaxed_Geometries_Intersect_Geometries-warm-long
143677 INFOGen  - Cache COLD
143677 INFOGen  -  Query 0
143677 INFOGen  -  Rep 0 <COMPLETED-NONE> 56349004 + 744380367 = 800729371 nsecs,
554 results, 0 scan errors
...
143677 INFOGen  - Cache WARM
143677 INFOGen  -  Query 0
143677 INFOGen  -  Rep 0 <COMPLETED-NONE> 364357 + 99903390 = 100267747 nsecs,
554 results, 0 scan errors
...
```

The run script starts by evaluating the arguments and deserializing any experiment related JSON specifications. Then, it performs system specific query translations and namespace prefix header merging. The experiment begins by executing the first iteration of the first query SC1 with COLD caches. The

full query to be executed is displayed, the system caches are cleared since a COLD cache execution has been requested and after a few second delay for Java garbage collection, the experiment launches an `RDF4JbasedExecutor` instance in a new thread for executing the query in a time constrained manner. The child process reports back every 6 hours (21600000 msecs) which is the 25% of the total query timeout (24 hours)[10] specified in the scalability execution specification. The executor reports all state transitions for the query execution and during the scanning phase reports the first 3 results of the resultset according to the `simplereportspec_original.json` logging specification. Before exiting, the executor reports that it completed with no errors (COMPLETED-NONE), the query evaluation, scanning and total time in nano seconds, the number of results in the resultset, the number of scan errors[11] and that the accuracy could not be verified, because the query-set specification provided did not include the expected number of results for this query. The collector then flushes in the report sink (PostgreSQL geographica3 database) the `2 cache types * 3 queries * 3 iterations = 18 results` and also generates the standard result and statistics files in the file system.

E. Evaluation Results

Figure 5 presents the evaluation results for the GeoRDFBench Framework. All four systems calculate the correct number of results for all queries. Overall, Strabon performs the best in all cache type and query combinations and exhibits a major improvement of warm cache performance over cold cache. Second best is RDF4J which performs predictably since warm cache times are faster than cold ones, response times for higher selectivity join SC3 are smaller than lower selectivity join SC2 which means that filters are considered early in the query plan and finally scales in an analog manner when repository size grows. GraphDB and Stardog share the third position but with different strengths and weaknesses. They share similar performance for SC1 with warm caches. GraphDB performs better in SC1 cold and SC3 and provides marginal improvements with warm caches if selectivity is not low. Stardog performs much better than GraphDB in the low selectivity spatial join SC2 which indicates that its data storage scheme performs better in full scans than GraphDB's storage scheme. Stardog warm cache times show concrete improvements over cold cache ones, but independent of cache type, performance degrades faster as repository size scales. This is consistent with the behavior of RocksDB [47] key-value database of Stardog, whose read overhead gets very high as the number of files (repository size) to be merged grows.

			GraphDB		RDF4J		Stardog		Strabon	
Cache	Query	Workload	Results	Time (sec)	Results	Time (sec)	Results	Time (sec)	Results	Time (sec)
Cold	SC1	10K	554	0.23	554	***0.18***	554	1.11	554	0.26
		100K	6278	0.69	6278	0.69	6278	1.66	6278	***0.45***
		1M	80500	3.39	80500	3.83	80500	6.43	80500	***3.32***
	SC2	10K	2	1.87	2	0.13	2	2.21	2	***0.09***
		100K	239	14.52	239	5.38	239	9.22	239	***0.47***
		1M	813	144.70	813	21.21	813	70.97	813	***1.83***
	SC3	10K	2	0.11	2	0.10	2	2.55	2	***0.03***
		100K	239	5.31	239	5.12	239	9.26	239	***0.39***
		1M	239	5.54	239	14.00	239	69.41	239	***1.87***
Warm	SC1	10K	554	0.10	554	0.12	554	0.15	554	***0.07***
		100K	6278	0.35	6278	0.35	6278	0.34	6278	***0.16***
		1M	80500	3.16	80500	3.25	80500	2.15	80500	***1.84***
	SC2	10K	2	1.55	2	0.08	2	0.85	2	***0.01***
		100K	239	14.65	239	5.15	239	6.71	239	***0.02***
		1M	813	145.47	813	18.94	813	63.77	813	***0.05***
	SC3	10K	2	0.07	2	0.08	2	1.18	2	***0.01***
		100K	239	5.21	239	5.07	239	7.06	239	***0.01***
		1M	239	5.66	239	11.50	239	63.80	239	***0.05***

Fig. 5: Scalability Workload Evaluation - Results & Times

The results of the above benchmark are available through Zenodo [48].

For the interested reader, the GeoRDFBench web site contains 4 Docker examples. The *"Multiple stores against multiple Scalability Workloads"* example is similar to the above evaluation as it engages RDF4J, GraphDB and JenaGeoSPARQL against the Scalability 10K, 100K, 1M workloads. The *"Multiple stores against LUBM(1,0) Workload"* example exhibits the use of GeoRDFBench Framework in running RDF4J, GraphDB and JenaGeoSPARQL against a SPARQL benchmark.

VII. 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 [42], [49], [50]. The first releases of GeoRDFBench and GeoRDFBench Samples are also available through Zenodo [46], [51].

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.

[10]The Scalability query timeout is very high because of the very high query response times in the 500M dataset.

[11]Usually due to invalid geometries or unsupported operators.

References

[1] E. C. J. R. Centre. (2007) INSPIRE Directive web site. [Online]. Available: https://inspire.ec.europa.eu/inspire-directive/2

[2] E. Commission. (2006) SWING Project web site. [Online]. Available: https://cordis.europa.eu/project/id/026514

[3] E. Commission. (2019) ExtremeEarth Project web site. [Online]. Available: http://earthanalytics.eu/

[4] Manolis Koubarakis, Ed., *Geospatial data science: a hands-on approach based on geospatial technologies*. ACM Books, 2023.

[5] O. Erling and I. Mikhailov, "Virtuoso: Rdf support in a native rdbms," in *Semantic web information management: a model-based perspective*. Springer, 2009, pp. 501–519.

[6] K. Kyzirakos, M. Karpathiotakis, and M. Koubarakis, "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*. Springer Berlin Heidelberg, 2012, pp. 295–311.

[7] D. Bilidas, T. Ioannidis, N. Mamoulis, and M. Koubarakis, "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*. Springer, 2022, pp. 411–427.

[8] R. Battle and D. Kolas, "Enabling the geospatial Semantic Web with Parliament and GeoSPARQL," *Semantic Web*, vol. 3, no. 4, pp. 355–370, 2012.

[9] M. Perry, A. Estrada, S. Das, and J. Banerjee, "Developing geosparql applications with oracle spatial and graph." in *SSN-TC/OrdRing@ ISWC*, 2015, pp. 57–61.

[10] T. Ioannidis, G. Garbis, K. Kyzirakos, K. Bereta, and M. Koubarakis, "Evaluating geospatial rdf stores using the benchmark geographica 2," *Journal on Data Semantics*, vol. 10, no. 3-4, pp. 189–228, 2021.

[11] N. Karalis, G. M. Mandilaras, and M. Koubarakis, "Extending the YAGO2 knowledge graph with precise geospatial knowledge," in *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part II*, ser. Lecture Notes in Computer Science, C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. F. Cruz, A. Hogan, J. Song, M. Lefrançois, and F. Gandon, Eds., vol. 11779. Springer, 2019, pp. 181–197. [Online]. Available: https://doi.org/10.1007/978-3-030-30796-7_12

[12] A. Dsouza, N. Tempelmeier, R. Yu, S. Gottschalk, and E. Demidova, "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*. ACM, 2021, pp. 4475–4484.

[13] K. Janowicz, P. Hitzler, W. Li, D. Rehberger, M. Schildhauer, R. Zhu, C. Shimizu, C. K. Fisher, L. Cai, G. Mai, J. Zalewski, L. Zhou, S. Stephen, S. G. Estrecha, B. D. Mecum, A. Lopez-Carr, A. Schroeder, D. Smith, D. J. Wright, S. Wang, Y. Tian, Z. Liu, M. Shi, A. D'Onofrio, Z. Gu, and K. Currier, "Know, know where, knowwheregraph: A densely connected, cross-domain knowledge graph and geo-enrichment service stack for applications in environmental intelligence," *AI Mag.*, vol. 43, no. 1, pp. 30–39, 2022. [Online]. Available: https://doi.org/10.1609/aimag.v43i1.19120

[14] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *Journal of Web Semantics*, vol. 3, no. 2-3, pp. 158–182, 2005.

[15] C. Bizer and A. Schultz, "The berlin sparql benchmark," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 5, no. 2, pp. 1–24, 2009.

[16] M. Morsey, J. Lehmann, S. Auer, and A.-C. Ngonga Ngomo, "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*. Springer, 2011, pp. 454–469.

[17] G. Garbis, K. Kyzirakos, and M. Koubarakis, "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*. Springer, 2013, pp. 343–359.

[18] 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 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 619–630.

[19] M. Saleem, Q. Mehmood, and A.-C. Ngonga Ngomo, "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*. Springer, 2015, pp. 52–69.

[20] W3C. (2013) SPARQL 1.1 Query Language web site. [Online]. Available: https://www.w3.org/TR/sparql11-query/

[21] Matthew Perry and John Herring, "OGC GeoSPARQL - A Geographic Query Language for RDF Data," Open Geospatial Consortium, OGC Implementation Standard OGC 11-052r4, Sep. 2012. [Online]. Available: http://www.opengis.net/doc/IS/geosparql/1.0

[22] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *Proceedings of the 2018 international conference on management of data*, 2018, pp. 1433–1445.

[23] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *Proceedings of the 15th Symposium on Database Programming Languages*. ACM, oct 2015. [Online]. Available: https://doi.org/10.1145%2F2815072.2815073

[24] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, "PGQL: a property graph query language," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, P. A. Boncz and J. L. Larriba-Pey, Eds. ACM, 2016, p. 7. [Online]. Available: https://doi.org/10.1145/2960414.2960421

[25] R. Angles, M. Arenas, P. Barceló, P. A. Boncz, G. H. L. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. F. Sequeda, O. van Rest, and H. Voigt, "G-CORE: A core for future graph query languages," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 1421–1432. [Online]. Available: https://doi.org/10.1145/3183713.3190654

[26] *Information technology - Database languages - GQL*, ISO/IEC Std. 39 075, April 2024. [Online]. Available: https://www.iso.org/standard/76120.html

[27] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized search trees for database systems," in *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, U. Dayal, P. M. D. Gray, and S. Nishio, Eds. Morgan Kaufmann, 1995, pp. 562–573. [Online]. Available: http://www.vldb.org/conf/1995/P562.PDF

[28] Oracle. (2019) Spatial And Graph web site. [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/19/spatial-and-graph.html

[29] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, pp. 1–9, 1974.

[30] Wikipedia. (2008) Geohash wiki page. [Online]. Available: https://en.wikipedia.org/wiki/Geohash

[31] K. Patroumpas, G. Giannopoulos, and S. Athanasiou, "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, 2014, pp. 301–310.

[32] T. Osman and G. Albiston, "Geosparql-jena: Implementation and benchmarking of a geosparql graphstore," in *23rd European Conference on Knowledge Management Vol 2*. Academic Conferences and publishing limited, 2022.

[33] P. Bellini and P. Nesi, "Performance assessment of RDF graph databases for smart city services," *J. Vis. Lang. Comput.*, vol. 45, pp. 24–38, 2018. [Online]. Available: https://doi.org/10.1016/j.jvlc.2018.03.002

[34] M. Jovanovik, T. Homburg, and M. Spasić, "A geosparql compliance benchmark," *ISPRS International Journal of Geo-Information*, vol. 10, no. 7, p. 487, 2021.

[35] F. Conrads, J. Lehmann, M. Saleem, M. Morsey, and A.-C. Ngonga Ngomo, "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*. Springer, 2017, pp. 48–65.

[36] H. Thakkar, "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*. Springer, 2017, pp. 256–266.

[37] C. Kostopoulos, G. Mouchakis, A. Troumpoukis, N. Prokopaki-Kostopoulou, A. Charalambidis, and S. Konstantopoulos, "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*. Springer, 2021, pp. 664–679.

[38] A. Troumpoukis, S. Konstantopoulos, G. Mouchakis, N. Prokopaki-Kostopoulou, C. Paris, L. Bruzzone, D.-A. Pantazi, and M. Koubarakis, "Geofedbench: A benchmark for federated geosparql query processors." in *ISWC (Demos/Industry)*, 2020, pp. 228–232.

[39] M. Röder, D. Kuchelev, and A. N. Ngomo, "HOBBIT: A platform for benchmarking big linked data," *Data Sci.*, vol. 3, no. 1, pp. 15–35, 2020. [Online]. Available: https://doi.org/10.3233/ds-190021

[40] A.-C. N. Ngomo, S. Auer, J. Lehmann, and A. Zaveri, "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.

[41] E. Jiménez-Ruiz, T. Saveta, O. Zamazal, S. Hertling, M. Roder, I. Fundulaki, A. N. Ngomo, M. Sherif, A. Annane, Z. Bellahsene *et al.*, "Introducing the hobbit platform into the ontology alignment evaluation campaign," in *13th International Workshop on Ontology Matching (OM)*, vol. 2288, 2018, pp. 49–60.

[42] T. Ioannidis. (2023) GeoRDFBench Framework web site. [Online]. Available: https://geordfbench.di.uoa.gr/

[43] MvnRepository. (2024) Maven Central Repository web site. [Online]. Available: https://mvnrepository.com/repos/central

[44] A. Owens, A. Seaborne, N. Gibbins *et al.*, "Clustered tdb: a clustered triple store for jena," 2008.

[45] FasterXML. (2024) FasterXML Jackson source code on Github. [Online]. Available: https://github.com/FasterXML/jackson

[46] T. Ioannidis. (2024, jul) GeoRDFBench Framework Samples v1.0.0 doi. [Online]. Available: https://doi.org/10.5281/zenodo.13120112

[47] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, S. H. Noh and B. Welch, Eds. USENIX Association, 2020, pp. 209–223. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[48] T. Ioannidis. (2024, aug) GeoRDFBench Framework Benchmark Results v1.0 doi. [Online]. Available: https://doi.org/10.5281/zenodo.13150398

[49] T. Ioannidis. (2023) GeoRDFBench Framework source code on Github. [Online]. Available: https://github.com/tioannid/geordfbench

[50] T. Ioannidis. (2023) GeoRDFBench Framework Samples source code on Github. [Online]. Available: https://github.com/tioannid/geordfbench_samples

[51] T. Ioannidis. (2024, jul) GeoRDFBench Framework v1.0.0 doi. [Online]. Available: https://doi.org/10.5281/zenodo.12666544