# The GeoRDFBench Framework: Geospatial semantic benchmarking simplified

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

Nikos Mamoulis
nikos@cs.uoi.gr
Department of Informatics and Telecommunications, University of Ioannina
Ioannina, Greece

Manolis Koubarakis
koubarak@di.uoa.gr
Department of Informatics and Telecommunications, National and Kapodistrian University of Athens
Athens, Greece

## Abstract

We present the GeoRDFBench framework, whose purpose is to assist and streamline the benchmarking of geospatial semantic stores. We identify and formally define all benchmark components, extend them to represent their geospatial aspects, allow for the automatic mapping of datasets to graphs, provide a specialization hierarchy of queryset types for micro or macro experimental scenarios, even for modeling dynamically generated queries. Queries may define their expected resultset to enable automatic accuracy verification. Experiment behavior and execution logic is controlled by the execution specification, which dictates the action (run experiment or print ground queries) to take, the number of repetitions per execution type (cold, warm, continuous cold), the query repetition and experiment timeouts, the delay period before clearing caches, the aggregating function for reporting execution times, and the policy to follow upon cold execution time out. We decouple these declarative benchmark specifications from the framework's execution engine and serialize them as JSON files; this way, we increase their reuse (instantiation through deserialization), experiment reproducibility and dissemination. We also model the Geospatial RDF store optional application and database server modules and manage their life-cycle (start, stop, restart) during experiment execution to achieve ideal cold cache query executions. In addition, we unify by generalization the repository and connection functionalities of the three most common RDF framework Java APIs offered by RDF stores: OpenRDF Sesame, Eclipse RDF4J and Apache Jena. At the same time GeoRDFBench allows queryset filtering, automatic system-dependent query namespace prefix generation and query rewriting when non GeoSPARQL spatial vocabularies are used. We provide for a quick learning start by implementing several geospatial RDF stores as separate runtime-dependent modules with repository generation and experiment execution scripts. RDF modules include: RDF4J with and without Lucene, GraphDB, Stardog, Strabon, OpenLink Virtuoso and Jena GeoSPARQL.

## Keywords

geospatial, semantic, benchmarking, framework

## 1 Introduction

Many projects [8, 10, 11] have shown that spatial and temporal aspects of Linked Open Data (LOD) are as important and critical, as thematic information, in order to guide decision making [40]. Graph database systems with varying degrees of spatial support have recently been used to manage very large LOD datasets [3, 5, 14, 16, 29, 30, 34, 37, 55]. Selecting the most suitable system requires *evaluating the most recent versions of available systems on accessible infrastructure within the defined budget, with the desired queryloads run against datasets of appropriate size and content.*

There have been many efforts on automating some of the arduous and repetitive tasks of benchmarking. Parametric ontology-based synthetic generators [6, 15, 20, 21, 42] have assisted in creating datasets of desired size and attributes, while log-mining techniques [15, 42, 63] 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 [59]. 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 geospatial benchmark experience on single node and Spark-based distributed RDF stores along with synthetic data and query generation [5, 20, 29] 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* with three main objectives.

First, it should *conceptualize and formalize geospatial semantic benchmarks and their execution environment making this knowledge as reusable as possible*, to enable users creating new benchmarks, modify existing ones in as fast, credible and repeatable way as possible.

Secondly, it should *model the RDF stores' architecture, the client RDF Java providers they offer and, on behalf of the system evaluators, implement as much of the configuration, optimization, module administration, data loading and query execution logic as possible* in order to minimize the effort of adding new RDF stores or upgrading existing ones for testing through the framework.

Thirdly, *the framework should be extensible*, meaning that, the benchmark components, such as querysets, datasets, execution model, the execution environment components, such as host, operating system, reporting sink, the RDF stores' architecture modules such as application and database servers and the client RDF Java providers should provide all commonly identified functionality and structural information but can be extended if needed in order to include new subclasses with modified behavior.

*Preprint on Zenodo, Athens, Greece*
2025. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM

Our work, at this stage, focuses on benchmarking single node GeoSPARQL/SPARQL query engines through the console. It makes it easy integrating new or updated RDF stores, provides a model for declaratively describing a benchmark's structure, behavior and its host's environment. It helps mapping existing benchmarks or designing new ones, it is aware of and assists with the spatial aspects of systems and benchmark workloads and provides support for GeoSPARQL/SPARQL query execution exception handling. It also allows parallel experiment execution of implemented stores in the same node with or without containers[1]. Perpendicular to our direction, HOBBIT [59] provides a generalized architecture with message bus connected containers for benchmarking the entire Linked Data (LD) life cycle and non LD workloads and systems. However this generality also forces it to be completely agnostic and therefore provide no assistance to its users for systems' integration, dealing with the spatial aspects of the benchmark workloads or handling the specific GeoSPARQL/SPARQL query language structure, features and execution issues.

To the best of our knowledge, there is no previous 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, operating systems 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 [29] benchmark's components, a PostgreSQL and an H2 [43] embedded implementation of the JDBC report sink and hosts with Linux and Windows operating systems*.
(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 [56] with or without Lucene SAIL [57], (ii) GraphDB [47], (iii) Stardog [64], (iv) Strabon [37], (v) Virtuoso [16], (vi) Jena GeoSPARQL [52]*. 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 and provides a *query translation hook*. 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*[3] 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 [21] benchmark's workload component is included in the JSON specification library.

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 explains GeoRDFBench's runtime. Showcasing the JSON generator API follows in section 5. Section 6 shows an evaluation of the framework and section 7 compares GeoRDFBench with HOBBIT and Geographica 2 benchmark. Section 8 presents conclusions and future work. Finally, the appendix features a comparison of GeoRDFBench to Geographica 2.

## 2 Related Work and Background

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

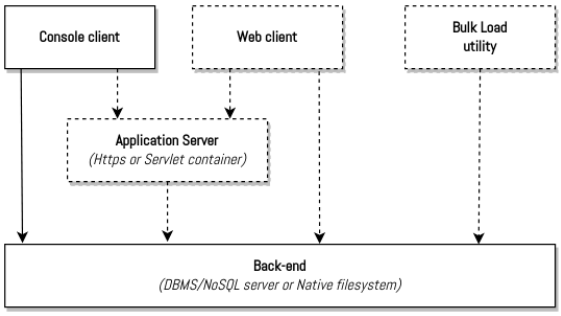
*Graph Store Categories.* Graph stores in general, follow either the Resource Description Framework (RDF) or the Labeled Property Graph (LPG) approach. RDF as a data model has good expressivity, while featuring a standardized declarative query language SPARQL [68] and a standardized spatial vocabulary GeoSPARQL [41]. 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 [19], Apache TinkerPop Gremlin [60], the Oracle supported PGQL [67] and G-Core [2] 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 [1] 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. Their architecture is similar to that in Figure 1 which shows their 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

[1] GeoRDFBench's site includes containerized images for demonstration purposes.
[2] GeoRDFBench framework is a Java Maven multi-module POM project.
[3] Usually due to invalid geometries or unsupported operators.

**Figure 1: Geospatial Graph Store Architectures**




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.

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 [3] uses the embedded key-value database Berkeley DB with a standard R-tree spatial index, Strabon [37] uses PostgreSQL and PostGIS with an R-tree over GiST [22] 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 [51] 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 it poses a critical challenge to the performance of these stores under query loads of interest [5, 36]. For spatial data, we face an additional challenge which is the approximate matching supported by the precision parameter of common spatial indexing algorithms, such as quad [18] and geohash [69] prefix trees, which basically controls how many results will match a spatial filter. Better accuracy requires more index storage and brings a storage and performance penalty, so most systems try to balance between the two. The spatial index algorithm and precision are either fixed for the store e.g., RDF4J, defined upon database creation e.g., Stardog or dynamically defined even after database creation e.g., GraphDB. Setting up a system with lower precision, has the benefit of reduced storage size, bulk load time and query execution times. Therefore, we have 4 important check points that a geospatial semantic benchmark should measure when testing spatially enabled stores: (i) *equality of spatial index precision* for all systems under test, (ii) *bulk load ability* for huge dataset sizes, (iii) *query execution performance* for various query loads and (iv) *query execution accuracy*. The most valid query execution accuracy test is comparing the query resultset against the expected resultset *(gold standard)*, as it is done in the Evaluation Storage module of HOBBIT. However storage requirements for persisting large gold standard 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[4], 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 execution accuracy since it is fast to verify and with low storage requirements which makes it both 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 [21], BSBM [6], DBpedia SPARQL benchmark (DBPSB) [42] and the Social Network Benchmark (SNB) [15], just to mention a few. GeoSPARQL benchmarks have been presented in [52, 54], the benchmark Geographica in [20], a smart city services related benchmark in [4], a compliance benchmark in [33] and Geographica 2 in [29].

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 assists with: (i) the integration of systems of interest, (ii) integration of existing benchmarks, (iii) 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.

[4] Removing problematic queries or reconfiguring the system.

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 [63] 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 [12] 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 [65] 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 [35] cloud benchmarking engine for federated query processors includes the GeoFedBench [66] benchmark, which focuses on validation of the actual crop land usage against the Austrian land survey dataset.

*Benchmarking Framework Platforms.* These are multi-user environments where researchers can store and share datasets, querysets, execution results and system modules. They are designed for deployment to cloud infrastructures, with distributed file systems and containerization technologies. HOBBIT [59], the most complete of these platforms, extends the scope of benchmarking to the entire LD life-cycle [45], such as *link discovery* [31] and allows the integration of systems in various programming languages. For a comparison between GeoRDFBench and HOBBIT please refer to Section 7.

## 3 GeoRDFBench: A Framework Simplifying Geospatial Semantic Benchmarking

In this section we present the high level architecture of the GeoRDFBench framework, which is shown in Figure 2.

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* 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.

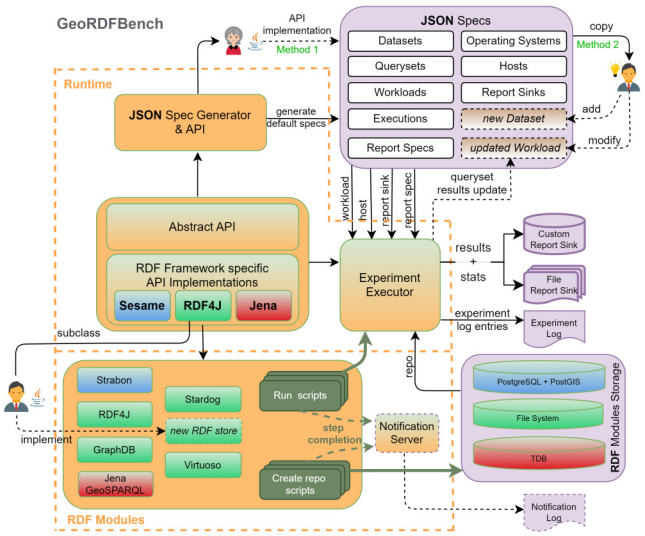



**Figure 2: Architectural overview of GeoRDFBench Framework**

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. The user can either implement new custom specifications using the JSON Specs Generator API (**Method 1**) or copy and modify existing ones (**Method 2**).

## 4 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.

### 4.1 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[5] Enhanced-ER (EER) diagrams, in Figure 3 and in a part of Figure 4, show the important entities, their specializations, how they are associated, along with the corresponding structural constraints (cardinality constraints) for these associations.

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

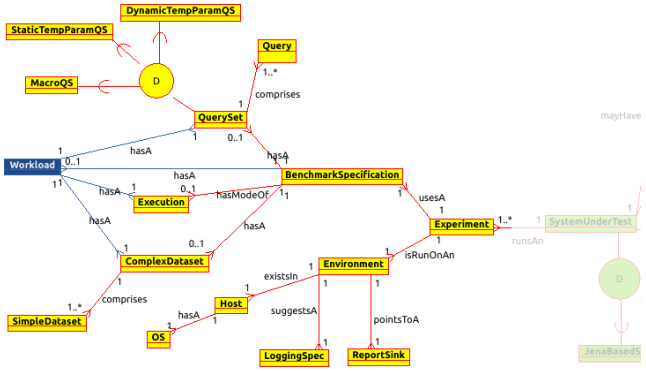



**Figure 3: EER diagram of Abstract API - Experiment Components**

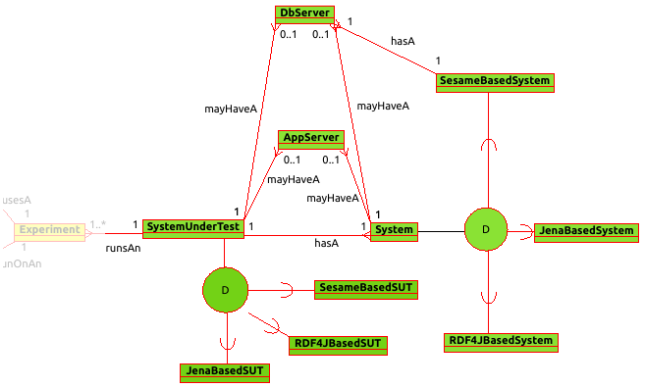



**Figure 4: EER diagram of Abstract API - System Under Test**

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.

*4.1.1 Experiment Components.* Figure 3 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 3, 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.

***Queryset.*** The entity `QuerySet`, in Figure 3, 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 inclusively or exclusively 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"[6] experiment types, the `MacroQS` subclass and its specializations[7] are useful for "macro"[8] experiment types.

***Execution specification.*** The entity `Execution`, in Figure 3, 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 3, 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 3, 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 3, 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

[6] Independent queries, each run several times with cold or warm caches.
[7] Not shown in Figure 3 for simplicity reasons.
[8] A sequence of queries representing a case scenario, that is run repeatedly as a whole.

repository files and the base path for the default reports and statistics.

***Report sink.*** The entity `ReportSink`, in Figure 3, 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. A second lighter option is the H2 embedded database which offers synchronous experiment result recording. In both cases, the target report sink database schema is generated with the help of the runtime-bundled *database generation SQL script*.

***Logging or Report specification.*** The entity `LoggingSpec`, in Figure 3, 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.

*4.1.2 Systems and SUTs.* In Figure 4, the parent concepts of system and system under test (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 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 5, 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.

## 4.2 RDF Framework Specific APIs

The second part of the core API, on the conceptual level, is depicted in part of Figure 4 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 5, and (ii) the *RDF Framework Specific SUT Layer*, which is the right side of the same level, both of which are explained below.

*4.2.1 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 4.3.15 was selected since is widely supported by all contemporary systems. 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
```

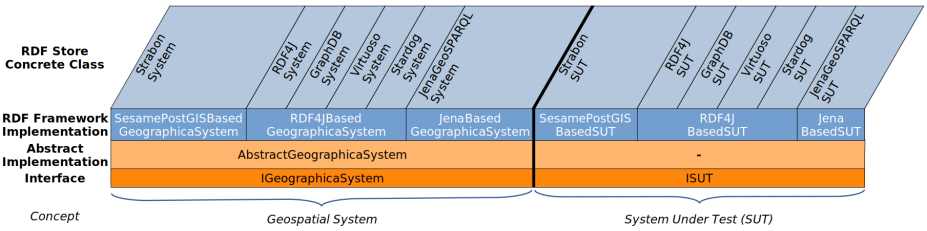



**Figure 5: Systems & SUT Class Hierarchies**

*Jena API* (`JenaBasedGeographicaSystem`). Only Jena GeoSPARQL uses this API, we simply chose a recent and frequently used Jena version in Maven Central [44] , from the latest stable branch, which was 4.10.0. Jena Tuple Database (TDB2) [53] was preferred as the persistent storage option as it is more robust and can handle large update transactions. 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`

*4.2.2 RDF Framework Specific SUT Layer.* This layer comprises three base implementations of the `ISUT` interface with 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 2) 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 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 [17] JSON library is used to annotate interfaces and class hierarchies to simplify serialization and deserialization.

# 5 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 1) is intended to evaluate geospatial RDF store scalability against increasingly larger real-world datasets with many complex geometries. There are 6 workload variants, each comprising one dataset with increasing number of triples (10K, 100K, 1M, 10M, 100M, 500M), a set with either 3 spatial function queries (1 selection and 2 joins) or with 3 equivalent spatial predicate queries[9] and a common experiment execution model. The 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. Listing 6 encodes all the above.

**Table 1: 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 | | |

## 5.1 Dataset generation

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

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

**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);
}
```

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}
```

## 5.2 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 55.927663412470 31,-5.940856933593749
          54.59116279530599,-3.1668090820312504 51.47967237816337,23.708496093749996
          37.95719224376526))\"^^<http://www.opengis.net/ont/geosparql#wktLiteral>",
   "FUNCTION" : "sfIntersects"
},
"mapGraphPrefixes" : { } }
```

## 5.3 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. Both methods of generating specifications are depicted in the upper part of Figure 2.

For an example of generating the *compact form* of a benchmark specification, the user can refer to *GeoRDFBench Framework Samples* [24] application that is using as test case, the LUBM benchmark for SPARQL.

## 6 Experimental Evaluation

In this section, we present the process of running some of the Geographica 2 [29] 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.

### 6.1 Environment

*Host Hardware.* The hardware platform for the experiments was an Intel NUC8i7BEH box, Ubuntu 22.04.5 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.

*Installed Software.* Virtuoso and GraphDB required that we install the corresponding server software under `/data`. In addition PostgreSQL v14.17 with PostGIS v3.2 was also installed, since Strabon requires it for creating spatial databases. The PostgreSQL "data_directory" was set in `/data/pgdata/data`. The branch `releases/2.0.0-M1` of GeoRDFBench was used in `/data/geordfbench`. The project came ready with prebuilt binaries. The report sink schema was left to be automatically created by the first system to run experiments. Manually creating the same schema is provided through the `scripts/geordfbench.sql` script.

*System-Version Selection.* Some RDF modules included in GeoRDFBench allow, without effort, testing system configuration variations. In view of this, we chose the following `seven (7)` systems and system variations to participate in this demonstration: (i) GraphDB `v10.8.5`, (ii) GraphDB `v10.8.5` with *GeoSPARQL plugin* enabled, *quad* prefix tree indexing algorithm and *11* precision value (iii) RDF4J `v4.3.15`, (iv) RDF4J `v4.3.15` with *Lucene Sail* enabled for spatial indexing (v) Openlink Virtuoso Open Server (VOS) `v7.2.14` (vi) Jena GeoSPARQL `v4.10.0` and (vii) Strabon `v3.3.3-SNAPSHOT`. Additionally, we tested the second GraphDB variant against two versions of the querysets: (i) with *spatial functions* which is the default for all systems and (ii) with *spatial predicates*, to exhibit the different optimization paths followed and the importance of testing all these alternatives offered for properly benchmarking spatially enabled RDF stores. We were not provided with a license renewal for Stardog[10] therefore we were unable to run experiments with the latest `v8.2.2` of the module which is included with GeoRDFBench. However, we do include it in the description of other aspects of the benchmarking process.

Since each GeoRDFBench RDF module is a Java client to an RDF store (embedded or standalone server) the current implementations use a Java RDF Framework library to instrument the stores. We currently have GraphDB, RDF4J and Virtuoso using *Eclipse RDF4J*, Jena GeoSPARQL uses *Apache Jena* while Strabon uses *OpenRDF Sesame*.

**Table 2: Scalability datasets basic characteristics**

| Dataset | # of Features | # of Points | # of Lines | # of Polygons |
|---|---|---|---|---|
| 10K | 1,135 | 587 | 0 | 900 |
| 100K | 12,166 | 6,623 | 4,239 | 2,531 |
| 1M | 118,161 | 46,781 | 45,238 | 29,200 |
| 10M | 1,038,739 | 317,865 | 328,630 | 427,842 |
| 100M | 10,259,959 | 904,677 | 2,058,386 | 7,553,440 |
| 500M | 48,623,878 | 5,520,767 | 15,771,932 | 23,390,220 |

### 6.2 Benchmark description

The following experiment aims both at studying performance characteristics of the SUTs but more importantly exhibiting the usage of GeoRDFBench for running benchmarks. We chose the 10K, 100K and 1M variants of the Scalability workload (see Table 1) where the respective datasets contain 10K, 100K and 1M triples. All share the same queryset which comprises three queries: (i) a spatial selection *SC1*, (ii) a low selectivity (heavy) spatial join *SC2* and (iii) a higher selectivity (lighter) spatial join *SC3*. The queryset is an instance of the `StaticTempParamQS` class with one template parameter, which represents a file persisted polygon replaced during queryset instantiation, used in the spatial filter of query SC1. The basic characteristics of the datasets (e.g., features and geometries) are described in Table 2.

The following listing shows the ground spatial function queryset for RDF4J:

**Listing 7: RDF4J Ground Spatial Function 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:
```

[10] Stardog requires a license otherwise experiments fail.

```
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)).
}
```

For comparison, the following listing shows the ground spatial predicate queryset for GraphDB+P:

**Listing 8: GraphDB Ground Spatial Predicate Queries & Prefix Header**

```
PREFIX ext: <http://rdf.useekm.com/ext#>
...
# Query 0 - SC1_Geometries_Intersects_GivenPolygon:
SELECT ?s1 ?o1 WHERE {
   ?s1 geo:asWKT ?o1 .
   ?s1 geo:sfIntersects "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 ?g1 ;
   lgo:has_code "1001"^^xsd:integer .
   ?s2 geo:hasGeometry ?g2 ;
   lgo:has_code ?code2 .
   ?g1 geo:sfIntersects ?g2 .
   FILTER(?code2>5000 && ?code2<6000 && ?code2 != 5260) .
}
# Query 2 - SC3_Relaxed_Geometries_Intersect_Geometries:
SELECT ?s1 ?s2 WHERE {
   ?s1 geo:hasGeometry ?g1 ;
   lgo:has_code "1001"^^xsd:integer .
   ?s2 geo:hasGeometry ?g2 ;
   lgo:has_code ?code2 .
   ?g1 geo:sfIntersects ?g2 .
   FILTER(?code2 IN (5622, 5601, 5641, 5621, 5661)) .
}
```

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 Virtuoso also include the following system-specific namespace prefixes:

**Listing 9: GraphDB & Virtuoso Prefix Header Differences**

```
# GraphDB adds a few useful GeoSPARQL extensions based on the USeekM library
PREFIX ext: <http://rdf.useekm.com/ext#>
# Access to Virtuoso's built-in functions
PREFIX bif: <http://www.openlinksw.com/schemas/bif#>
```

To produce the ground queries of Listings 7 and 8 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 10: 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.

## 6.3 Experiment Executions

As explained, we have 7 systems to test, however two variants of the scalability queryset need to run against the repositories of the GeoSPARQL enabled GraphDB. Therefore there is a total of `eight (8)` different system configurations to be tested against three increasingly bigger workloads. Since each system configuration needs to run against the three scalability workloads (10K, 100K, 1M) there are `twenty four (24)` experiment executions with unique database ID, system configuration name and configuration description are depicted in Table 3.

**Table 3: System Configuration Experiments**

| ExpIDs | ConfigName | System | ConfigDesc |
|---|---|---|---|
| 1-3 | **RDF4J** | RDF4J | v4.3.15 |
| 4-6 | **JenaGeoSPARQL** | Jena GeoSPARQL | v4.10.0 |
| 7-9 | **GraphDB** | GraphDB | v10.8.5 |
| 10,11,14 | **Virtuoso** | Virtuoso Open Server | v7.2.14, *2D Rtree* |
| 15-17 | **GraphDB+** | GraphDB + *GeoSPARQL plugin* | v10.8.5, *Quad-11* |
| 18-20 | **GraphDB+P** | GraphDB + *GeoSPARQL plugin* | v10.8.5, *Quad-11, Pred* |
| 21-23 | **RDF4J+** | RDF4J + *Lucene Sail* | v4.3.15, *Lucene spatial index* |
| 31-33 | **Strabon** | Strabon | 3.3.3-SNAPSHOT |

GeoRDFBench provides the following common tools: (i) the **environment preparation script** *geordfbench/scripts/prepareRunEnvironment.sh* which prepares the environment variables for running the RDF Module specific scripts for a given host, (ii) the **print environment script** *geordfbench/scripts/printRunEnvironment.sh* which allows reviewing the environment variables previously set.

Each RDF Module provides the following scripts which have largely the same functionality: (i) the **repository generation wrapper script** which creates all enabled repositories, imports RDF data into them and spatially indexes them, (ii) the **compact workload run script** which tests the system against the compact representation of a workload and (iii) the **detailed workload run script** which tests the system against the detailed representation of a workload.

The repository generation wrapper scripts require that the user either provides all the required arguments for the system at hand and host critical locations, or that the environment variables have

already been prepared with the environment preparation script, in which case the single required boolean parameter denotes whether to overwrite existing repositories or not. 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 to them.

For the purpose of this work and to exhibit the use of these tools we created a single very simple **experiment execution script** for each one of the eight required executions. They basically call upon the GeoRDFBench common and the system-specific scripts in the correct order. They have an almost identical functionality with few minor additions for special cases. The log files generated by these scripts are three types: (i) `envvars_<sys>.log` which record the environment variables reported by the print environment script, (ii) `logCreateRepos_Scal10K_1M_<sys>.log` which store the repository generation wrapper script output and (iii) `RunWL<sys>Exp_Scal10K.log` which record the compact or detailed workload run script output. Below we see a compact (comment and whitespace sanitized) version of RDF4J's script:

**Listing 11: RDF4J Experiment execution script**

```
#!/bin/bash
CWD=`pwd`
cd /data/geordfbench/scripts
# environment preparation script sets environment variables for RDF4JSUT and nuc8i7beh host
source prepareRunEnvironment.sh nuc8i7beh RDF4JSUT "CreateRepo_Scalability10K_1M_RDF4J"
# print environment script logs environment variables for RDF4JSUT and nuc8i7beh host
./printRunEnvironment.sh >> /data/envvars_rdf4j.log

BASE_LOG_DIR=/data/LOGS
REPO_CREATION_LOG_DIR=${BASE_LOG_DIR}/RepoCreation/${ActiveSUT}
EXP_RUN_LOG_DIR=${BASE_LOG_DIR}/ExperimentRun/${ActiveSUT}
mkdir -p ${REPO_CREATION_LOG_DIR}; mkdir -p ${EXP_RUN_LOG_DIR}

cd ../RDF4JSUT/scripts/CreateRepos/
# Enable 100K and 1M scalability dataset generation
sed -i -e 's@^levels=( "10K" )@levels=( "10K" "100K" "1M" )@g' createAllRDF4JRepos.sh
# repository generation wrapper script creates repos, loads data, spatial indexs them
./createAllRDF4JRepos.sh false 2>&1 | tee -a
        ${REPO_CREATION_LOG_DIR}/logCreateRepos_Scal10K_1M_RDF4J.log

DateTimeISO8601=`date --iso-8601='date'` # Record current date
cd ../RunTests3/
# compact workload run script runs RDF4J against the Scalability 10K workload
./runWLTestsForRDF4JSUT.sh -Xmx24g \
  -rbd ${RDF4JRepoBaseDir//"${EnvironmentBaseDir}/"} -expdesc
         ${DateTimeISO8601}_RDF4JSUT_RunWL_Scal10K \
  -wl ${GeoRDFBenchJSONLibDir}/workloads/scalabilityFunc10K_WLoriginal_GOLD_STANDARD.json \
  -h ${GeoRDFBenchJSONLibDir}/hosts/nuc8i7behHOSToriginal.json \
  -rs ${GeoRDFBenchJSONLibDir}/reportspecs/simplereportspec_original.json \
  -rpsr ${GeoRDFBenchJSONLibDir}/reportsources/nuc8i7behHOSToriginal.json 2>&1 | tee -a
          ${EXP_RUN_LOG_DIR}/RunWLRDF4JExp_Scal10K.log
# compact workload run script runs RDF4J against the Scalability 100K workload
# compact workload run script runs RDF4J against the Scalability 1M workload
# ..
```

Excluding file and path locations, the difference of the experiment execution script for RDF4J compared to RDF4J+ variant is the line enabling the Lucene Sail, as shown below:

**Listing 12: diff(RDF4J, RDF4J+) Experiment execution scripts**

```
$ diff -U 1 runRDF4J_Scal10K_1M.sh runRDF4J_Lucene_Scal10K_1M.sh
...
@@ -4,4 +4,5 @@
source prepareRunEnvironment.sh nuc8i7beh RDF4JSUT "CreateRepo_Scalability10K_1M_RDF4J"
+ export EnableLuceneSail=true
./printRunEnvironment.sh >> /data/envvars_rdf4j.log
...
```

The difference between the GraphDB and GraphDB+ is also a single line enabling the GeoSPARQL plugin which could require 2 more assignments if a different spatial algorithm and/or accuracy were needed.

**Listing 13: diff(GraphDB, GraphDB+) Experiment execution scripts**

```
$ diff -U 1 runGraphDB_Scal10K_1M.sh runGraphDB_GeoSPARQL_Scal10K_1M.sh
...
@@ -3,2 +3,5 @@
source prepareRunEnvironment.sh nuc8i7beh GraphDBSUT "CreateRepo_Scalability10K_1M_GraphDB"
+ export EnableGeoSPARQLPlugin=true
+ # export IndexingAlgorithm = quad
+ # export IndexingPrecision = 11
./printRunEnvironment.sh >> /data/envvars_graphdb.log
...
```

## 6.4 Experiment log files

*6.4.1 Environment variables log.* There is one such log for each system configuration. It has a homogeneous output layout which comprises two sections. The first one contains environment variables common for all systems, such as, host name, IP address and max memory, locations for dataset source files, standard output results and GeoRDFBench installation-related. The second section consists of system-specific variables such as, version number, server installation (if present) and repositories' locations. Below we see part of the JenaGeoSPARQL output:

**Listing 14: JenaGeoSPARQL Environment variables**

```
All SUTs
--------
Environment = NUC8I7BEH
EnvironmentBaseDir = /data
GeoRDFBenchScriptsDir = /data/geordfbench/scripts
GeoRDFBenchJSONLibDir = /data/geordfbench/json_defs
DatasetBaseDir = /data/Geographica2_Datasets
...
ResultsDirName = 531f9c1#_2025-04-26_JenaGeoSPARQLSUT_CreateRepo_Scalability10K_1M_JenaGeoSPARQL
ActiveSUT = JenaGeoSPARQLSUT
ExperimentDesc = 531f9c1#_2025-04-26_JenaGeoSPARQLSUT_CreateRepo_Scalability10K_1M_JenaGeoSPARQL
SystemMemorySizeInGB = 32 GBs
JVM_Xmx = -Xmx24g

JenaGeoSPARQL SUT
-----------------
JenaBaseDir = /data/apache-jena-4.10.0
JenaGeoSPARQLRepoBaseDir = /data/JenaGeoSPARQL_4.10.0_Repos
Version = 4.10.0
```

*6.4.2 Repository generation log.* There is one such log for each system configuration. It has the least homogeneous output since each system's preferred loader uses significantly different strategies which are reflected in the output. Apart from very useful details provided which may allow for fine-tuning each system, it provides two key pieces of information which are the *final repository size* and the *required time to complete repository creation, data loading and data indexing* for all enabled repositories. Below we see part of the RDF4JSUT+ output for the *scalability_10K* repository:

**Listing 15: RDF4JSUT+ repository generation log**

```
...
Running script with syntax:
./createAllRDF4JRepos.sh false /data/Geographica2_Datasets
        /data/RDF4J_4.3.15_LuceneRepos/server -Xmx24g true 192.168.1.44 3333
        /data/Geographica2_Datasets/Scalability/scalabilityDSGen.sh
        /data/Geographica2_Datasets/Scalability/scalability500MRefDS.nt.gz
Script start time: Mon 28 Apr 2025 04:21:18 pm EEST
/data/RDF4J_4.3.15_LuceneRepos/server dir does not exist. Creating it now ...
Checking/Creating scalability 10K dataset ... Scalability 10K dataset already exists
Generating scalability 10K repository ...
./createRDF4JRepo.sh /data/RDF4J_4.3.15_LuceneRepos/server scalability_10K false "spoc,posc"
        N-TRIPLES /data/Geographica2_Datasets/Scalability/10K -Xmx24g true
        "http://www.opengis.net/ont/geosparql#asWKT" 192.168.1.44 3333
CREATE_REPO_ARGS = createman "/data/RDF4J_4.3.15_LuceneRepos/server" "scalability_10K" "FALSE"
        "true" "spoc,posc" "http://www.opengis.net/ont/geosparql#asWKT"
LOAD_REPO_ARGS = dirloadman "/data/RDF4J_4.3.15_LuceneRepos/server" "scalability_10K"
        "N-TRIPLES" "/data/Geographica2_Datasets/Scalability/10K" true
0    [main] INFO RDF4JSystem - No LocalRepositoryManager instance present, creating a new one.
68   [main] INFO RDF4JSystem - Creating NativeStore base sail with spoc,posc indexes and
         forceSync = false
71   [main] INFO RDF4JSystem - Adding Lucene sail on top of NativeStore
73   [main] INFO RDF4JSystem - Lucene sail will spatially index properties:
         http://www.opengis.net/ont/geosparql#asWKT
```

```
216  [main] INFO RDF4JSystem - Creating new repository object for repo id = scalability_10K
775  [main] INFO RDF4JSystem - Initialing new repository object for repo id = scalability_10K
835  [main] INFO RepoUtil - RDF4J created with manager lucene repo
       "/data/RDF4J_4.3.15_LuceneRepos/server/repositories/scalability_10K" in 74 msecs
835  [main] INFO RDF4JSystem - Closing connection...
837  [main] INFO RDF4JSystem - Repository closed.
1    [main] INFO RDF4JSystem - Loading file scalability_10K.nt ...
2500 [main] INFO RDF4JSystem - Finished loading file scalability_10K.nt in 2492 msecs
2506 [main] INFO RepoUtil - RDF4J loaded with manager all files from
       "/data/Geographica2_Datasets/Scalability/10K" to repo
       "/data/RDF4J_4.3.15_LuceneRepos/server/repositories/scalability_10K" in 2599 msecs
2506 [main] INFO RDF4JSystem - Closing connection...
2508 [main] INFO RDF4JSystem - Repository closed.
RDF4J repository "/data/RDF4J_4.3.15_LuceneRepos/server/repositories/scalability_10K" has size:
       7MB
```

*6.4.3 Run workload log.* Each such log concerns a single run of a system against a workload. It has a quite homogeneous output layout which comprises three sections. In the first section all arguments are listed and validated, the JSON specifications are deserialized to create the workload, host and system related components, the system query timeout is set dynamically based on the execution specification, if needed the queryset is translated according to the system and the final namespace prefix header is formed by merging dataset, queryset and system dependent prefixes. Below we see a sanitized version of the first section for Virtuoso's run against the Scalability 1M workload:

**Listing 16: Virtuoso run log start section for Scalability 1M**

```
...
0    [main] INFO RunVirtuosoExperimentWorkload - args[0] = -surl
1    [main] INFO RunVirtuosoExperimentWorkload - args[1] = http://localhost:1111
1    [main] INFO RunVirtuosoExperimentWorkload - args[2] = -susr
1    [main] INFO RunVirtuosoExperimentWorkload - args[3] = dba
1    [main] INFO RunVirtuosoExperimentWorkload - args[4] = -spwd
1    [main] INFO RunVirtuosoExperimentWorkload - args[5] = dba
1    [main] INFO RunVirtuosoExperimentWorkload - args[6] = -rbd
2    [main] INFO RunVirtuosoExperimentWorkload - args[7] = virtuoso-opensource-7.2.14/repos
2    [main] INFO RunVirtuosoExperimentWorkload - args[8] = -expdesc
2    [main] INFO RunVirtuosoExperimentWorkload - args[9] = 2025-04-27_VirtuosoSUT_RunWL_Scal1M
2    [main] INFO RunVirtuosoExperimentWorkload - args[10] = -wl
2    [main] INFO RunVirtuosoExperimentWorkload - args[11] =
       /data/geordfbench/json_defs/workloads/scalabilityFunc1M_WLoriginal.json
2    [main] INFO RunVirtuosoExperimentWorkload - args[12] = -h
2    [main] INFO RunVirtuosoExperimentWorkload - args[13] =
       /data/geordfbench/json_defs/hosts/nuc8i7behHOSToriginal.json
2    [main] INFO RunVirtuosoExperimentWorkload - args[14] = -rs
2    [main] INFO RunVirtuosoExperimentWorkload - args[15] =
       /data/geordfbench/json_defs/reportspecs/simplereportspec_original.json
2    [main] INFO RunVirtuosoExperimentWorkload - args[16] = -rpsr
2    [main] INFO RunVirtuosoExperimentWorkload - args[17] =
       /data/geordfbench/json_defs/reportsources/nuc8i7behHOSToriginal.json
5    [main] INFO RunVirtuosoExperimentWorkload - |==> Experiment related options
9    [main] INFO RunVirtuosoExperimentWorkload - Experiment description:
       2025-04-27_VirtuosoSUT_RunWL_Scal1M
9    [main] INFO RunVirtuosoExperimentWorkload - |==> Workload, Host, Report related options
9    [main] INFO RunVirtuosoExperimentWorkload - Workload specs configuration JSON file:
       /data/geordfbench/json_defs/workloads/scalabilityFunc1M_WLoriginal.json
10   [main] INFO RunVirtuosoExperimentWorkload - List of queries to include in the run: all
10   [main] INFO RunVirtuosoExperimentWorkload - Host configuration JSON file:
       /data/geordfbench/json_defs/hosts/nuc8i7behHOSToriginal.json
10   [main] INFO RunVirtuosoExperimentWorkload - Report specs configuration JSON file:
       /data/geordfbench/json_defs/reportspecs/simplereportspec_original.json
11   [main] INFO RunVirtuosoExperimentWorkload - Report source specs configuration JSON file:
       /data/geordfbench/json_defs/reportsources/nuc8i7behHOSToriginal.json
11   [main] INFO RunVirtuosoExperimentWorkload - |==> Repository/Store options
11   [main] INFO RunVirtuosoExperimentWorkload - Virtuoso Server endpoint URL:
       http://localhost:1111
11   [main] INFO RunVirtuosoExperimentWorkload - Virtuoso server username: dba
12   [main] INFO RunVirtuosoExperimentWorkload - Virtuoso server password: dba
12   [main] INFO RunVirtuosoExperimentWorkload - |==> System options
12   [main] INFO RunVirtuosoExperimentWorkload - BaseDir: virtuoso-opensource-7.2.14/repos
284  [main] INFO RunVirtuosoExperimentWorkload - {SimpleGeospatialWL: ScalabilityFunc, ,
{GeographicaDS: scalability_1M, Scalability/1M,
{GenericGeospatialSimpleDS: scalability_1M, N-TRIPLES}}
{StaticTempParamQS: scalabilityFunc, ,false, SimpleES{ COLD=3, WARM=3, action=RUN,
       maxduration=604800 secs, repmaxduration=86400 secs, func=QUERY_MEDIAN }}}
590  [main] INFO RDF4JBasedSUT - Reading VirtuosoSUT properties from file :
       jar:file:/data/geordfbench/VirtuosoSUT/target/VirtuosoSUT-2.0.0-SNAPSHOT.jar!/virtuoso.properties
603  [main] INFO VirtuosoSUT - Initializing..
603  [main] INFO VirtuosoSUT - Starting Virtuoso server...
15634 [main] INFO VirtuosoSystem - Uninitialized VirtuosoRepository created with query timeout
       = 0
15634 [main] INFO VirtuosoSystem - Initialized VirtuosoRepository has query timeout = 86400
...
15713 [main] INFO ExperimentWorkload - VirtuosoSystem-dependent translation of the queryset
       scalabilityFunc
15713 [main] INFO ExperimentWorkload - VirtuosoSystem-dependent namespace prefixes merged with
       the prefixes of queryset scalabilityFunc
15714 [main] INFO VirtuosoSUT - Closing..
15715 [main] INFO VirtuosoSystem - Closing connection...
15715 [main] INFO VirtuosoSystem - Repository closed.
20728 [main] INFO VirtuosoSUT - Stopping Virtuoso server...
```

In the second section, we find the details of each query execution step. It 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)[11] 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 `args(15) = 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[12] and that the accuracy could not be verified, because the queryset specification provided did not include the expected number of results for this query. In Listing 18 we see a sanitized version of the second iteration of the execution of query SC2 (no 1) with COLD caches against the Scalability 100K workload.

**Listing 17: Virtuoso run log SC2 COLD Scalability 100K**

```
115549 [main] INFO ExperimentWorkload - |==> Executing query [1,
       SC2_Intensive_Geometries_Intersect_Geometries] (COLD, 1):
PREFIX bif: <http://www.openlinksw.com/schemas/bif#>
...
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

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)).
}
...
120969 [main] INFO UbuntuJammyOS - Caches cleared after delay of 5000 msecs
120970 [main] INFO VirtuosoSUT - Starting Virtuoso server...
133318 [main] INFO VirtuosoSystem - Initialized VirtuosoRepository has query timeout = 86400
133352 [main] INFO VirtuosoSUT - Starting QueryExecutor thread
133353 [main] INFO VirtuosoSUT - Waiting for QueryExecutor thread to finish
133353 [main] INFO VirtuosoSUT - Timeout progress step is 21600000 msecs
133353 [Thread-5] INFO QueryRepResult - Transitioning (NOTSTARTED => STARTED)
133354 [Thread-5] INFO QueryRepResult - Transitioning (STARTED => PREPARING)
133355 [Thread-5] INFO QueryRepResult - Transitioning (PREPARING => EVALUATING)
135404 [Thread-5] INFO QueryRepResult - Transitioning (EVALUATING => EVALUATED)
135405 [Thread-5] INFO RDF4JbasedExecutor -  s1  s2
135405 [Thread-5] INFO RDF4JbasedExecutor - ------------------------------------>
135405 [Thread-5] INFO QueryRepResult - Transitioning (EVALUATED => SCANNING)
135405 [Thread-5] INFO RDF4JbasedExecutor -
       http://data.linkedeodata.eu/osm/wales/places/id/335184
       http://data.linkedeodata.eu/osm/wales/transport/id/123360606
135405 [Thread-5] INFO RDF4JbasedExecutor -
       http://data.linkedeodata.eu/osm/wales/places/id/335184
       http://data.linkedeodata.eu/osm/wales/transport/id/123360628
135405 [Thread-5] INFO RDF4JbasedExecutor -
       http://data.linkedeodata.eu/osm/wales/places/id/335184
       http://data.linkedeodata.eu/osm/wales/transport/id/1396066034
135766 [Thread-5] INFO QueryRepResult - Transitioning (SCANNING => SCANNED)
135766 [Thread-5] INFO RDF4JbasedExecutor - <------------------------------------
135766 [Thread-5] INFO QueryRepResult - Transitioning (SCANNED => COMPLETED)
135766 [main] INFO VirtuosoSUT - Percentage of expired timeout is 0.0 %
135776 [main] INFO ExperimentWorkload - |<== Executed query [1,
       SC2_Intensive_Geometries_Intersect_Geometries] (COLD, 1): <COMPLETED-NONE> 2050015972 +
       360751232 = 2410767204 nsecs, 239 results, 0 scan errors - ACCURACY NOT DETERMINED
```

[11] The Scalability query timeout is very high because of the very high query response times in the 500M dataset.
[12] Usually due to invalid geometries or unsupported operators.

In the third section, we have the result collector's actions which involves flushing the [2 cache types * 3 queries * 3 iterations = 18] cached results in the report sink (PostgreSQL writes in *geordf-bench* database in deferred mode, while in H2 results are written in synchronous mode). The collector also generates the standard result and statistics files in the file system and finally reports the gross script run time. A reduced version of the Virtuoso's run against the Scalability 10K workload is shown below:

**Listing 18: Virtuoso run log Scalability 10K results**

```
289581 [main] INFO PostgreSQLRepSrc - Deferred mode for PostgreSQLRepSrc was enabled. 18
        records were flushed
289582 [main] INFO GenericExprerimentResultsCollector - Export statistics in
"/data/Results_Store/VirtuosoSUT/2025-04-27_VirtuosoSUT_RunWL_Scal10K/
Scalability/10K/VirtuosoSUT-ExperimentWorkload"
289740 [main] INFO GenericExprerimentResultsCollector - Created non existing directory
289760 [main] INFO GenericExprerimentResultsCollector - Statistiscs printed:
        ./00-SC1_Geometries_Intersects_GivenPolygon-cold
289772 [main] INFO GenericExprerimentResultsCollector - Statistiscs printed:
        ./00-SC1_Geometries_Intersects_GivenPolygon-cold-long
...
289777 [main] INFO GenericExprerimentResultsCollector - Statistiscs printed:
        ./02-SC3_Relaxed_Geometries_Intersect_Geometries-warm
289777 [main] INFO GenericExprerimentResultsCollector - Statistiscs printed:
        ./02-SC3_Relaxed_Geometries_Intersect_Geometries-warm-long
289778 [main] INFO GenericExprerimentResultsCollector - Cache COLD
289778 [main] INFO GenericExprerimentResultsCollector -  Query 0
289779 [main] INFO GenericExprerimentResultsCollector -   Rep 0 <COMPLETED-NONE> 634086428 +
        114910498 = 748996926 nsecs, 554 results, 0 scan errors
289779 [main] INFO GenericExprerimentResultsCollector -   Rep 1 <COMPLETED-NONE> 579085856 +
        123844201 = 702930057 nsecs, 554 results, 0 scan errors
289779 [main] INFO GenericExprerimentResultsCollector -   Rep 2 <COMPLETED-NONE> 444039862 +
        123960680 = 568000542 nsecs, 554 results, 0 scan errors
289779 [main] INFO GenericExprerimentResultsCollector -  Query 1
...
289780 [main] INFO GenericExprerimentResultsCollector -  Query 2
...
289780 [main] INFO GenericExprerimentResultsCollector - Cache WARM
289780 [main] INFO GenericExprerimentResultsCollector -  Query 0
...
289780 [main] INFO GenericExprerimentResultsCollector -  Query 1
289780 [main] INFO GenericExprerimentResultsCollector -   Rep 0 <COMPLETED-NONE> 158021881 +
        383204 = 158405085 nsecs, 2 results, 0 scan errors
289780 [main] INFO GenericExprerimentResultsCollector -   Rep 1 <COMPLETED-NONE> 157411100 +
        355432 = 157766532 nsecs, 2 results, 0 scan errors
289780 [main] INFO GenericExprerimentResultsCollector -   Rep 2 <COMPLETED-NONE> 157869590 +
        304380 = 158173970 nsecs, 2 results, 0 scan errors
289780 [main] INFO GenericExprerimentResultsCollector -  Query 2
...
289781 [main] INFO RunVirtuosoExperimentWorkload - End ScalabilityFunc
Start time = Sun 27 Apr 2025 07:45:49 pm EEST
End time = Sun 27 Apr 2025 07:50:39 pm EEST
```

## 6.5 Evaluation Results

Tables 4-6 present the evaluation results for the experiments run with GeoRDFBench Framework.

*6.5.1 Repository size.* In terms of repository size, RDF4J's *Native-store* [58] (B-Tree indexed, no spatial index) performs best in small to medium dataset sizes and quite well for millions of triples which is on a par with its specification. In the second place we have Jena-GeoSPARQL's *TDB2* and RDF4J+ which exhibit top performance. For JenaGeoSPARQL this is no surprise as the 3 spatial indexes are persisted in memory. RDF4J+ albeit having a *Lucene* quad spatial index enabled, it keeps a very small storage footprint which actually becomes the smallest as input dataset size increases. GraphDB and its variant GraphDB+ are storage efficient for larger datasets following the RDF4J and RDF4J+ but seem to have substantial overhead for small to medium ones. The former observation is quite logical since GraphDB is built on top of RDF4J and its *GeoSPARQL plugin* uses the same Lucene library as RDF4J+ for spatial indexing. Strabon's storage footprint is the second largest due to the *PostGIS* database with an *R-tree over GiST* [22] spatial index. Simirarly, a *2D Rtree* spatial index is used by Virtuoso which features the largest repository sizes, more than twice as big as the next system in any dataset size.

**Table 4: Scalability Workload Evaluation - Repository sizes (MB)**

| System ConfigName | 10K triples | 100K triples | 1M triples |
|---|---|---|---|
| RDF4J | 4 | 14 | 135 |
| JenaGeoSPARQL | 5 | 22 | 207 |
| GraphDB | 28 | 38 | 144 |
| Virtuoso | 77 | 133 | 521 |
| GraphDB{+,+P} | 32 | 44 | 201 |
| RDF4J+ | 7 | 21 | 205 |
| Strabon | 24 | 52 | 349 |

*6.5.2 Ingestion time.* In terms of data loading and indexing time RDF4J is again the fastest, followed by JenaGeoSPARQL and RDF4J+. Looking at the relative increase in time between the medium and large dataset we get a clear view that JenaGeoSPARQL's overall time is less sensitive to dataset size increase which is indicative of good scalability. The same can be said to an even larger degree for GraphDB's *Preload* [49] offline tool. This two-phase strategy bulk loader has a large overhead, evident in the small and medium datasets, but performs increasingly better as dataset sizes grow. Virtuoso's bulk loader shares the very good scalability of GraphDB's loader but not to the same degree. Strabon's three phase[13] bulk loader, *StrabonLoader*, greatly improves the data loading and indexing time over the standard OpenRDF Sesame based loader, however lags far behind the other systems. Its basic characteristic is that ingestion time is proportional to the dataset size and that is mainly due to the time spent in phase one, RDF to CSV, which is performed with the Redland raptor parser and serialization library.

**Table 5: Scalability Workload Evaluation - Data ingestion times (sec)**

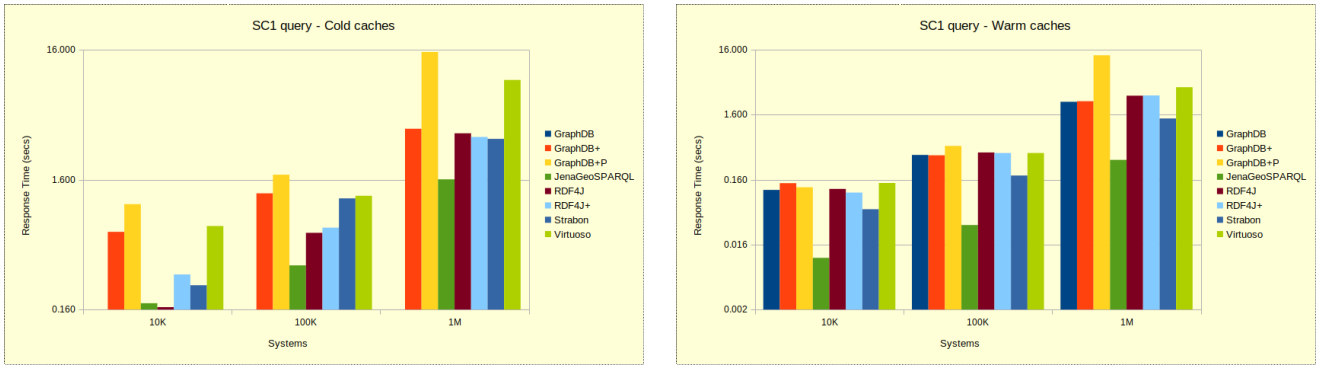
| System ConfigName | 10K triples | 100K triples | 1M triples |
|---|---|---|---|
| RDF4J | 0.62 | 1.80 | 11.14 |
| JenaGeoSPARQL | 2.49 | 3.91 | 16.61 |
| GraphDB | 14.89 | 11.82 | 17.59 |
| Virtuoso | 15.87 | 17.88 | 28.07 |
| GraphDB{+,+P} | 44.82 | 48.03 | 59.13 |
| RDF4J+ | 2.60 | 4.41 | 23.63 |
| Strabon | 20.42 | 50.91 | 287.32 |

*6.5.3 Accuracy & Response time.* Table 6 shows the number of results returned and response times of the eight systems for each one of the workload, query and cache type combinations. Figures 6-8 show chart diagrams with the system response times for the three queries. All results were accurately calculated by all systems with one exception. For the spatial selection query SC1, for the Scalability 1M workload, Virtuoso calculated **80501** instead of 80500 results returned by all other systems. This is probably due to some rounding error in the 2D Rtree spatial indexing process and/or in the implementation of `geof:sfIntersects` GeoSPARQL function which mishandles a border result.

[13] RDF to CSV, CSV to PostGIS tables, indexing tables

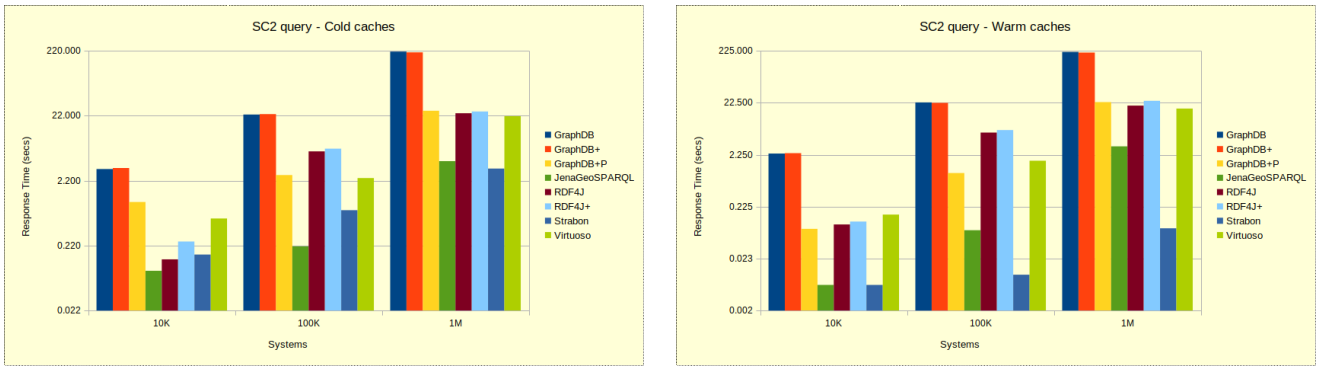
**Table 6: Scalability Workload Evaluation - Accuracy & Response Times**

| | | | RDF4J | | JenaGeoSPARQL | | GraphDB | | Virtuoso | | GraphDB+ | | GraphDB+P | | RDF4J+ | | Strabon | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Cache** | **Query** | **Workload** | **# Res** | **Time(s)** | **# Res** | **Time(s)** | **# Res** | **Time(s)** | **# Res** | **Time(s)** | **# Res** | **Time(s)** | **# Res** | **Time(s)** | **# Res** | **Time(s)** | **# Res** | **Time(s)** |
| **Cold** | **SC1** | 10K | 554 | 0.167 | 554 | 0.179 | 554 | 0.710 | 554 | 0.703 | 554 | 0.634 | 554 | 1.036 | 554 | 0.298 | 554 | 0.246 |
| | | 100K | 6278 | 0.623 | 6278 | 0.350 | 6278 | 1.294 | 6278 | 1.202 | 6278 | 1.255 | 6278 | 1.747 | 6278 | 0.683 | 6278 | 1.148 |
| | | 1M | 80500 | 3.639 | 80500 | 1.609 | 80500 | 3.918 | **80501** | 9.359 | 80500 | 3.943 | 80500 | 15.376 | 80500 | 3.408 | 80500 | 3.295 |
| | **SC2** | 10K | 2 | 0.135 | 2 | 0.090 | 2 | 3.320 | 2 | 0.575 | 2 | 3.439 | 2 | 1.031 | 2 | 0.254 | 2 | 0.160 |
| | | 100K | 239 | 6.209 | 239 | 0.214 | 239 | 22.907 | 239 | 2.411 | 239 | 23.256 | 239 | 2.682 | 239 | 6.838 | 239 | 0.771 |
| | | 1M | 813 | 23.995 | 813 | 4.392 | 813 | 213.015 | 813 | 21.658 | 813 | 207.580 | 813 | 26.186 | 813 | 25.539 | 813 | 3.373 |
| | **SC3** | 10K | 2 | 0.124 | 2 | 0.053 | 2 | 0.732 | 2 | 0.580 | 2 | 0.697 | 2 | 1.404 | 2 | 0.206 | 2 | 0.154 |
| | | 100K | 239 | 6.136 | 239 | 0.183 | 239 | 8.816 | 239 | 2.415 | 239 | 8.802 | 239 | 6.238 | 239 | 6.719 | 239 | 0.695 |
| | | 1M | 239 | 15.347 | 239 | 4.072 | 239 | 8.943 | 239 | 21.663 | 239 | 8.715 | 239 | 117.763 | 239 | 17.546 | 239 | 3.474 |
| **Warm** | **SC1** | 10K | 554 | 0.115 | 554 | 0.010 | 554 | 0.111 | 554 | 0.142 | 554 | 0.141 | 554 | 0.122 | 554 | 0.101 | 554 | 0.056 |
| | | 100K | 6278 | 0.419 | 6278 | 0.032 | 6278 | 0.384 | 6278 | 0.411 | 6278 | 0.379 | 6278 | 0.529 | 6278 | 0.411 | 6278 | 0.185 |
| | | 1M | 80500 | 3.136 | 80500 | 0.322 | 80500 | 2.524 | **80501** | 4.229 | 80500 | 2.577 | 80500 | 13.122 | 80500 | 3.160 | 80500 | 1.399 |
| | **SC2** | 10K | 2 | 0.102 | 2 | 0.007 | 2 | 2.364 | 2 | 0.158 | 2 | 2.405 | 2 | 0.084 | 2 | 0.116 | 2 | 0.007 |
| | | 100K | 239 | 5.991 | 239 | 0.079 | 239 | 22.781 | 239 | 1.717 | 239 | 22.300 | 239 | 0.997 | 239 | 6.689 | 239 | 0.011 |
| | | 1M | 813 | 19.755 | 813 | 3.252 | 813 | 213.212 | 813 | 17.335 | 813 | 207.504 | 813 | 23.082 | 813 | 24.454 | 813 | 0.086 |
| | **SC3** | 10K | 2 | 0.101 | 2 | 0.007 | 2 | 0.110 | 2 | 0.161 | 2 | 0.111 | 2 | 0.416 | 2 | 0.105 | 2 | 0.006 |
| | | 100K | 239 | 6.089 | 239 | 0.080 | 239 | 7.989 | 239 | 1.718 | 239 | 7.976 | 239 | 4.803 | 239 | 6.613 | 239 | 0.011 |
| | | 1M | 239 | 11.303 | 239 | 3.288 | 239 | 8.086 | 239 | 17.437 | 239 | 8.085 | 239 | 116.961 | 239 | 15.621 | 239 | 0.078 |




**(a)** **(b)**

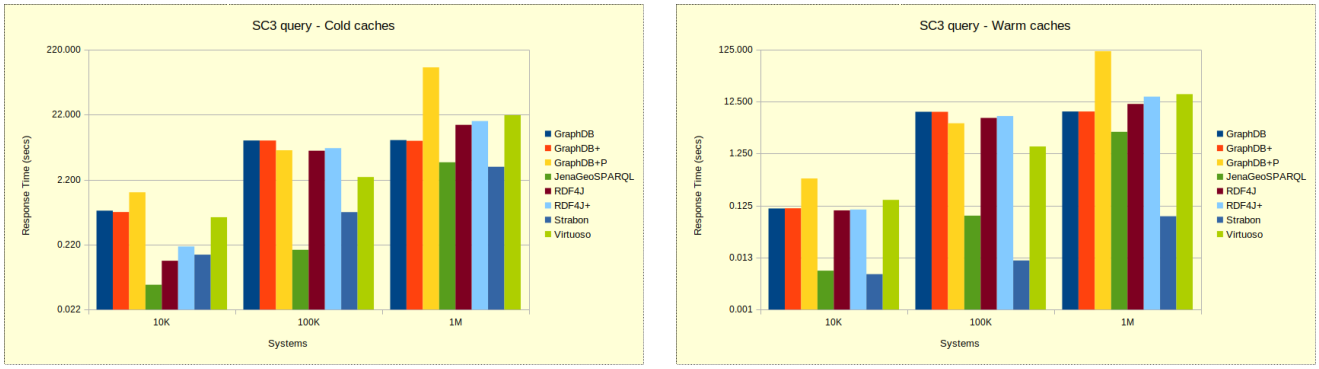
**Figure 6: Response times - SC1 spatial selection query**




**(a)** **(b)**

**Figure 7: Response times - SC2 spatial join query**

The response times are the median values out of three executions. JenaGeoSPARQL is the most performant system for COLD cache setting and spatial selection queries independent of caching. Strabon is very strong for WARM cache executions and particularly strong on spatial join queries regardless of caching. RDF4J is very fast with the smallest workload on COLD caches but we note that it performs predictably through out the entire spectrum. Warm cache times are on average 15% 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 response times scale in an analog manner when repository size grows. RDF4J+ provided minimal improvements over RDF4J only in the spatial selection but its overall query performance does not justify the extract repository size and additional ingestion time. Virtuoso performs adequately but its behaviour is not entirely predictable. While response times scale in an analog




**(a)** **(b)**

**Figure 8: Response times - SC3 spatial join query**

manner when repository size grows and warm cache times are on average 20% faster than cold ones, the response times for higher selectivity join SC3 are similar to the ones for lower selectivity join SC2 which means that it has a low selectivity threshold for engaging filters early in the query plan.

GraphDB exhibited two characteristics, lack of sensitivity to caching when selectivity is not low and slow response times for spatial joins especially the heavy SC2 query. Enabling the Lucene-based GeoSPARQL plugin and using the standard spatial function querysets, the GraphDB+ offered minimal benefits to the worst scenarios with larger datasets. Using the spatial predicate version of the querysets, as instructed in the official documentation [48], made possible the effective use of the GeoSPARQL plugin's spatial index and drastically improved the GraphDB+P variant's SC2 response times by a factor of **x10**. However this negatively affected the times for SC1 and SC3 queries, in some cases by a factor of **(x5)**. This variant also improved its sensitivity to caching.

The results of the above benchmark are available at a Github [28] and Zenodo [27] for verification.

## 6.6 Benchmarking with Docker - Examples

The GeoRDFBench web site [26] contains detailed tutorials and 4 Docker examples. The *"Multiple stores against multiple Scalability Workloads"* example engages RDF4J, GraphDB and JenaGeoSPARQL against the Scalability 10K, 100K, 1M workloads and exhibits features, such as: (i) customization (**Method 2**) of queryset specification, (ii) query accuracy checking and (iii) custom sink reporting. In the *"Multiple stores against LUBM(1,0) Workload"* example GeoRDFBench tests RDF4J, GraphDB and JenaGeoSPARQL against the LUBM(1,0) SPARQL benchmark workload and exhibits features, such as: (i) generation (**Method 1**) of workload compact specification, (ii) query inclusion filtering, (iii) query accuracy checking and (iv) SPARQL benchmark compatibility. The *"Strabon and Virtuoso against the Synthetic Workload"* example engages Strabon and Virtuoso against the Geographica 2 *Synthetic-512* workload and exhibits features, such as: (i) query inclusion filtering, (ii) customization (**Method 2**) of execution specification, (iii) query timeout handling and (iv) custom sink reporting. All the above are shown in more detail in Table 7.

# 7 Comparisons of GeoRDFBench

In this section we compare both the Geographica 2 benchmark which featured some aspects of a benchmarking framework and the HOBBIT benchmarking platform against GeoRDFBench Framework and showcase new features and improvements that are introduced by it.

## 7.1 GeoRDFBench vs HOBBIT

HOBBIT [59] is a very powerful but equally generic benchmarking framework. It *uses LD technologies, such as ontologies* to describe properties and parameters of benchmarks, systems and evaluation metrics and *uses OpenLink Virtuoso RDF store* as the platform's persistent storage. However the benchmarks and systems tested can be related to other subject areas distinct from LD, such as machine learning (ML) applications [62]. At the same time it *does not provide any special assistance for benchmarking the performance of linked data loading to and SPARQL query execution against RDF stores.* Also, *no provision for GeoSPARQL is present anywhere in the framework's APIs [23] or configuration.* GeoRDFBench, on the other hand, was specifically designed for benchmarking RDF data loading and GeoSPARQL/SPARQL query execution using the Java language. In addition to the above, it is our view *that the two frameworks target different user groups.*

The *average HOBBIT user* (according to designers' expectations) will be satisfied with reusing benchmarks and integrated systems in the form of containers provided by other *advanced HOBBIT users* and will simply modify the values of the models' exposed parameters. The premise is that there is such a **"group of the willing"** users that will be creating the relevant ontologies and containers for benchmarks and systems at the desired computer languages and updating containers when systems under test get newer versions. In reality the average HOBBIT user does not seem interested in embarking upon extending or modifying the relevant ontologies and containers of these benchmarks and systems and when they do so it is usually in the context of European funded sponsored challenges and mostly on the topic of Instance Matching or Link Discovery[14]. For example, to rerun experiments with a newer version of an existing system would at the very least require:

[14] https://hobbit-project.github.io/challenge_overview.html

**Table 7: Docker Examples Feature Matrix**

| | | | | | | |
|---|---|---|---|---|---|---|
| **Example** | Name | | **Scalability 10K Workload With RDF4J** | **RDF4J, GraphDB and JenaGeoSPARQL against Scalability-{10K, 100K, 1M} Workloads** | **Strabon and Virtuoso against The Synthetic Workload** | **RDF4J, GraphDB and JenaGeoSPARQL against LUBM(1, 0) Workload** |
| | Type | | Docker | Docker | Docker | Docker |
| | Host OS | | Linux | Windows 10 | Linux | Windows 10 |
| | Docker | *Image* | Manual Build | Pre-build from Github Registry | Pre-build from Github Registry | Pre-build from Github Registry |
| | | *Simulated Host* | NUC8i7BEH | NUC8i7BEH | NUC8i7BEH | NUC8i7BEH |
| **Experiment** | Startup Script | | Automatic with Container Start | Manual Script Start | Manual Script Start | Manual Script Start |
| **Benchmark** | Name | | Geographica 2 | Geographica 2 | Geographica 2 | LUBM |
| | Workload | *Representation* | Compact | Compact | Detailed | Compact |
| | | *Name* | Scalability | Scalability | Synthetic | LUBM(1, 0) |
| | | *Dataset 1* | 10K | 10K GOLD STANDARD | Scaling Factor N=512 | Scaling Factor N=1, GOLD STANDARD |
| | | *Dataset 2* | | 100K | | |
| | | *Dataset 3* | | 1M | | |
| **RDF Stores** | 1 | | RDF4J | RDF4J | Strabon | RDF4J |
| | 2 | | | GraphDB | Virtuoso | GraphDB |
| | 3 | | | JenaGeoSPARQL | | JenaGeoSPARQL |
| **Features** | *Result Accuracy Check* | | | **Yes** | | **Yes** |
| | *Query Inclusion Filter* | | | | **Yes** | **Yes** |
| | *Specification Customization* | | | **Yes** *(Queryset)* | **Yes** *(Execution)* | |
| | *JSON Generator API* | | | | | **Yes** |
| | *Auto Query Prefix Header* | | **Yes** | **Yes** | **Yes** | **Yes** |
| | *System Query Translation* | | | | **Yes** | |
| | *Query Timeout Handling* | | | | **Yes** | |
| | *Custom Sink Reporting* | | | **Yes** | **Yes** | **Yes** |
| | *GeoSPARQL Benchmarking* | | **Yes** | **Yes** | **Yes** | |
| | *SPARQL Benchmarking* | | | | | **Yes** |

(i) updating and rebuilding the source code of the *System Adapter* module and (ii) rebuilding the container[15] which includes it.

The *GeoRDFBench Framework's user* is mainly interested in benchmarking specifically RDF store performance in terms of data loading, data indexing, query accuracy and query execution times. They have expertise in Java programming and want to find each RDF store's optimal configuration for each host's capabilities, try alternative data loading methods for different dataset sizes, experiment with different spatial indexing algorithms and accuracies offered by each geospatial RDF store, have control over query execution in order to investigate exceptions either because of timeout expiration or the presence of malformed data or functionality compliance issues.

As such, it is our view *that the two frameworks are not competitive* and a direct comparison with quantitative criteria is neither possible nor useful. However, the qualitative comparison between the two frameworks that follows might help potential users find which one better fits their specific project' needs and provide ideas for improving both frameworks.

HOBBIT has been used to benchmark non-LD applications and, among other things, allows it to run benchmarks throughout the LD life-cycle [32, 62]. This flexibility is achieved by containerizing the benchmarking components and using the RabbitMQ [39] message broker, which allows *byte array* "messages" to be exchanged between them. This *stripped from semantics representation* is exactly what powers the general applicability of the HOBBIT framework, but *at the same time it carries a heavy price for the component developers who are obliged to take care all the application domain related critical tasks* without assistance from the platform APIs [23]. This functionality reusability at the container level also *creates dependencies of system containers' source code to modifications of the other ontologies*. For example, if new features are added to/removed from a benchmark's ontology, it not only triggers source code modification in the corresponding container but it also requires that all system adapter containers' source code needs to be aware of the new exposed benchmark features if one would like to test these systems against the updated version of the test. The *HOBBIT Java SDK* [61] provides some evidence [16] of the difficulties working with HOBBIT, such as: (i) demands a lot of reading, (ii) requires manual and error-prone work, (iii) does not allow to debug locally, etc. The HOBBIT Java SDK attempts to alleviate some of these problems.

For the application domain of "benchmarking GeoSPARQL enabled RDF stores" the developer has to acquire knowledge and implement tasks, such as: (i) RDF Store back-end choice with options, e.g., RDF4J+Lucene, (ii) setup optimized configurations (sizing memory resources to various components) which may differ for data load and query execution or when big data are involved, e.g., Virtuoso Performance Tuning [50], (iii) choose between normal and bulk data loaders when available (see GraphDB LoadRDF and Preload bulk load utilities), (iv) enable/build a geospatial index for a repository upon creation for some systems, e.g., Strabon, and Stardog, while for other systems, such as GraphDB [48] after data loading through a special plugin's operations, (v) choosing spatial index method and accuracy, e.g., GraphDB (vi) tagging GeoSPARQL spatial properties so that RDF Stores can perform spatial operations, e.g., RDF4J+Lucene [57], (vii) build namespace prefix header for each queryset usually needs to merge prefixes from the input dataset, the queryset itself and system dependent prefixes that define required/specialized vocabularies (viii) create a declarative mapping between files and contexts when triples from different origin are loaded in separate graphs (see Geographica 2 Real-world workload), (ix) query execution must be granular, able to handle exceptions though out its phases while being able to receive partial resultsets upon timeouts or gracefully skip results as in the case of "bad geometries".

For all the above and much more GeoRDFBench provides support through: (i) its runtime APIs, (ii) bundled functionality and

[15] https://hobbit-project.github.io/system_integration.html#1-set-up-the-system-adapter-project

[16] https://github.com/hobbit-project/java-sdk-example/blob/master/SDK_vs_Standard_Way.pdf

**Table 8: Feature Comparison with HOBBIT**

| Category | Feature | HOBBIT | GeoRDFBench |
|---|---|---|---|
| *Generic Features* | Programming Language support | **Java, Python,…** | **Java** |
| | Distributed System support | **yes** | **no** |
| | SUT Query Language support | **don't care** *(no support)* | **GeoSPARQL, SPARQL** |
| | Functionality (code) Reusability | **yes***(containers)* | **yes***(Runtime APIs, Scripts)* |
| | Structure (data) Reusability | **yes***(ontologies)* | **yes***(JSON specs, Scripts)* |
| | Metadata Model | **ontology** | **ER diagram** |
| *Assistance Testing RDF Stores -SPARQL/ GeoSPARQL Processors* | SUT Configuration optimization | **no** | **yes** |
| | Spatial Index setup | **no** | **yes** |
| | Query Namespace Header management | **no** | **yes***(auto)* |
| | Dynamic Queryset filtering | **no** | **yes***(Inc/Exclusive)* |
| | Dataset To Context mapping | **no** | **yes** |
| | WARM Cache Query exec | **yes** | **yes** |
| | COLD Cache Query exec | **no** | **yes** |
| | Query Timeout handling | **no** | **yes** |
| | Query Exception handling | **no** | **yes** |
| | RDF Framework Providers | - | **OpenRDF Sesame, RDF4J, Apache Jena** |

properties implemented as hierarchies of classes, (iii) JSON Generator utility classes and (iv) scripts that organize the configuration intricacies of prebundled systems and timely execution of critical actions when appropriate for each system. *Effectively, the researcher interested in SPARQL and especially GeoSPARQL benchmarking receives great help with the demanding parts of the task.* Table 8, summarizes the features of both systems.

## 7.2 GeoRDFBench vs Geographica 2

Geographica 2 benchmark is a set of well designed geospatial workloads and a set of RDF stores running against them following a specific set of steps and rules. However several areas were identified where much was left to expect for improvement. Abstraction and encapsulation [7, 9] were limited, leading to increased duplication of code and effort, especially when introducing new systems with a different RDF Java framework or with backward incompatible updated RDF Java framework version. Generalization-Specialization [7, 9] was limited to workload-specific queryset and experiment class hierarchies. Query namespace prefix header was constructed per dataset-system combination. Duplication of similar code could be found in the system setup, repository construction, query execution logic, statistics collection, etc. Host related parameters, such as maximum available system memory, mount point paths for repository storage, log and results collection, hard disk types, such as SSD vs magnetic disks, affected the configuration and optimization of the systems under test, therefore, making it very difficult and error prone running the same experiments in different hosts. In general, the structure of benchmark components (datasets, queryset, execution model) and experiment environment (host RAM, cache clearing method, locations for datasets and repositories) were not formally specified, were tightly coupled with the code base and the problem was only partially alleviated by multiple experiment execution parameters.

GeoRDFBench expands the scope of Universe of Discourse (UoD) [9] modeling to include all benchmark and experiment environment components, declaratively defines their specifications, decouples these from code by serializing them to disk, provides APIs and utilities to generate custom specifications and allows easy component creation from existing ones. Of paramount importance is the generalization of the repository and connection functionality of common RDF frameworks which allows significant reusability of code and easy integration of new systems with similar (in case of version upgrade) or different RDF frameworks.

As further quantitative evidence of the difference between Geographica 2 and GeoRDFBench a comparison is made between the two using the well established source code quality metric and effort predictor, the *Source Lines of Code (SLOC)* [38, 46]. More specifically, the utility **cloc** [13] measured the number of files and the *physical* SLOCs for the runtime and RDF modules for the two systems.

Table 9 presents the SLOC comparison between the two. The number of files and physical SLOCs of the GeoRDFBench runtime are **x2.54** and **x4.82** times the corresponding Geographica 2 ones. These numbers confirm the large scale of redesign and scope expansion that took place and which increased the opportunities for code reuse for RDF module implementation. This statement is consistent with the **reduction by half** of the number of physical SLOCs for the four common RDF modules: Strabon, RDF4J, GraphDB and Virtuoso. The newly introduced, Jena GeoSPARQL, required the least number of SLOCs because of its full GeoSPARQL compliance, while the newly introduced Stardog required the most effort since 50% of its SLOCs dealt with query translation to handle non-compliance to the GeoSPARQL standard. Also Stardog and Virtuoso required additional code to handle the server aspects of their architecture. RDF4Js SLOCs are relatively large because it encompasses the "dual" implementation of NativeStore with and without Lucene SAIL for spatial indexing.

Table 10 presents details on the Geographica 2 and GeoRDFBench classes supporting each feature/concept.

# 8 Conclusions and Future Work

We presented the concepts, architecture and basic operation of the GeoRDFBench Framework, which aims to: (i) save the geospatial semantic benchmark researcher's time and effort testing systems against benchmarks, (ii) minimize the margin for errors, (iii) increase reproducibility and results' verification, while (iv) remaining extensible. Source code, running examples and instructions are provided in our repositories [24, 25] and site [26].

Future work will include: (i) a *user interface module* for assisting with generating and modifying JSON specifications, (ii) an *endpoint module* for CRUD operations on JSON specification libraries, (iii) support for the *Hadoop file system* and (iv) a *fourth Spark-based framework API*, so that Spark-based distributed GeoSPARQL solutions, such as **Strabo 2** [5] can be tested.

# References


[1] ISO/IEC 2024. *Information technology - Database languages - GQL.* ISO/IEC. https://www.iso.org/standard/76120.html

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. 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*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1421–1432. doi:10.1145/3183713.3190654

[3] Robert Battle and Dave Kolas. 2012. Enabling the geospatial Semantic Web with Parliament and GeoSPARQL. *Semantic Web* 3, 4 (2012), 355–370.

[4] Pierfrancesco Bellini and Paolo Nesi. 2018. Performance assessment of RDF graph databases for smart city services. *J. Vis. Lang. Comput.* 45 (2018), 24–38.

**Table 9: SLOC Comparison with Geographica 2**

| **RDF Java Framework** | | | **OpenRDF Sesame** | | **RDF4J** | | | | | | | | **Jena** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **GeoRDFBench Module** | **Runtime** | | **Strabon SUT** | | **RDF4J SUT** | | **GraphDB SUT** | | **Virtuoso SUT** | | **Stardog SUT** | | **JenaGeoSPARQL SUT** | |
| **CLOC Indicator** | Files | Code Lines | Files | Code Lines | Files | Code Lines | Files | Code Lines | Files | Code Lines | Files | Code Lines | Files | Code Lines |
| **Geographica 2** | **31** | **2249** | 2 | **333** | 3 | **835** | 3 | **584** | 4 | **511** | - | - | - | - |
| **GeoRDFBench** | **79** | **10842** | 4 | **218** | 5 | **357** | 4 | **170** | 4 | **319** | 4 | **435** | 4 | **133** |
| *Increace (%)* | *155%* | *382%* | 100% | *-35%* | 67% | *-57%* | 33% | *-71%* | 0% | *-38%* | | | | |

**Table 10: Feature Comparison with Geographica 2**

| **Module** | **Feature / Concept Group** | **Geographica 2** | **GeoRDFBench** |
|---|---|---|---|
| **Run-time** | Supported RDF Framework | OpenRDF Sesame<br>RDF4J<br>- | OpenRDF Sesame<br>RDF4J<br>Jena |
| | Benchmark Concept To Class | QueriesSet* | <<IQuery>>, SimpleQuery<br><<IQuerySet>>, SimpleQS*<br><<IQuerySetPartOfWorkload>>, SimpleQSPartOfWorkload*<br><<ISimpleDataSet>>, <<IGeospatialSimpleDataSet>>, GenericGeospatialSimpleDS<br><<IGeospatialDataSet>>, GeographicaDS<br><<IExecutionSpec>>, SimpleES<br><<IGeospatialWorkLoadSpec>>, SimpleGeospatialWL |
| | Environment Concept To Class | - | <<IHost>>, SimpleHost<br><<IOS>>, GenericLinuxOS**<br><<IReportSource>>, JDBCRepSrc, PostgreSQLRepSrc, EmbeddedJDBCRepSrc, H2EmbeddedRepSrc<br><<IReportSpec>>, SimpleReportSpec |
| | System Concept To Class | <<SUT>><br>RunSUT | <<ISUT>>, SesamePostGISBasedSUT, RDF4JBasedSUT, JenaBasedSUT<br>RunSUTExperiment,<br>RunSUTExperimentWorkload<br><<IGeographicaSystem>>,<br>AbstractGeographicaSystem, SesamePostGISBasedGeographicaSystem,<br><br>RDF4JBasedGeographicaSystem, JenaBasedGeographicaSystem<br>Experiment, ExperimentWorkload |
| | Experiment Concept To Class | Experiment* | Experiment, ExperimentWorkload<br><<IExperimentResultsCollector>>, GenericExprerimentResultsCollector |
| | Utility To Class | - | QuerySetUtil, DataSetUtil, ExecutionSpecUtil, WorkLoadSpecUtil<br>HostUtil, ReportSourceUtil, ReportSpecUtil |
| * Set of classes which are workload-specific<br>** Set of classes for specific Linux variants | | **Bold** + << >> :<br>**Bold** + *Italic* :<br>**Bold** : | interface<br>abstact class<br>concrete class |


doi:10.1016/j.jvlc.2018.03.002

[5] Dimitris Bilidas, Theofilos Ioannidis, Nikos Mamoulis, and Manolis Koubarakis. 2022. 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, 411–427.

[6] Christian Bizer and Andreas Schultz. 2009. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)* 5, 2 (2009), 1–24.

[7] Grady Booch, Robert A Maksimchuk, Michael W Engle, Bobbi J Young, Jim Connallen, and Kelli A Houston. 2008. Object-oriented analysis and design with applications. *ACM SIGSOFT software engineering notes* 33, 5 (2008), 29–29.

[8] European Commission's Joint Research Centre. 2007. *INSPIRE Directive web site*. https://inspire.ec.europa.eu/inspire-directive/2

[9] Peter Coad, Edward Yourdon, et al. 1992. *Object-oriented analysis*. Vol. 2. Yourdon press New York.

[10] European Commission. 2006. *SWING Project web site*. https://cordis.europa.eu/project/id/026514

[11] European Commission. 2019. *ExtremeEarth Project web site*. http://earthanalytics.eu/

[12] Felix Conrads, Jens Lehmann, Muhammad Saleem, Mohamed Morsey, and Axel-Cyrille Ngonga Ngomo. 2017. 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, 48–65.

[13] Albert Danial. 2021. *cloc: v1.92 source code on Github*. https://github.com/AlDanial/cloc

[14] Alishiba Dsouza, Nicolas Tempelmeier, Ran Yu, Simon Gottschalk, and Elena Demidova. 2021. 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, 4475–4484.

[15] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[16] Orri Erling and Ivan Mikhailov. 2009. Virtuoso: RDF support in a native RDBMS. In *Semantic web information management: a model-based perspective*. Springer, 501–519.

[17] FasterXML. 2024. *FasterXML Jackson source code on Github.* https://github.com/FasterXML/jackson

[18] Raphael A Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4 (1974), 1–9.

[19] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*. 1433–1445.

[20] George Garbis, Kostis Kyzirakos, and Manolis Koubarakis. 2013. 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, 343–359.

[21] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics* 3, 2-3 (2005), 158–182.

[22] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. 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*, Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio (Eds.). Morgan Kaufmann, 562–573. http://www.vldb.org/conf/1995/P562.PDF

[23] HOBBIT. 2017. *HOBBIT General API.* https://hobbit-project.github.io/platform_api.html

[24] Theofilos Ioannidis. 2023. *GeoRDFBench Framework Samples source code on Github.* https://github.com/tioannid/geordfbench_samples

[25] Theofilos Ioannidis. 2023. *GeoRDFBench Framework source code on Github.* https://github.com/tioannid/geordfbench

[26] Theofilos Ioannidis. 2023. *GeoRDFBench Framework web site.* https://geordfbench.di.uoa.gr/

[27] Theofilos Ioannidis. 2025. *GeoRDFBench Framework Benchmark Results doi.* https://doi.org/10.5281/zenodo.15349539

[28] Theofilos Ioannidis. 2025. *GeoRDFBench Framework Benchmark Results on Github.* https://github.com/tioannid/geordfbench_samples

[29] Theofilos Ioannidis, George Garbis, Kostis Kyzirakos, Konstantina Bereta, and Manolis Koubarakis. 2021. Evaluating geospatial RDF stores using the benchmark Geographica 2. *Journal on Data Semantics* 10, 3-4 (2021), 189–228.

[30] Krzysztof Janowicz, Pascal Hitzler, Wenwen Li, Dean Rehberger, Mark Schildhauer, Rui Zhu, Cogan Shimizu, Colby K. Fisher, Ling Cai, Gengchen Mai, Joseph Zalewski, Lu Zhou, Shirly Stephen, Seila Gonzalez Estrecha, Bryce D. Mecum, Anna Lopez-Carr, Andrew Schroeder, Dave Smith, Dawn J. Wright, Sizhe Wang, Yuanyuan Tian, Zilong Liu, Meilin Shi, Anthony D'Onofrio, Zhining Gu, and Kitty Currier. 2022. 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 (2022), 30–39. doi:10.1609/aimag.v43i1.19120

[31] Ernesto Jiménez-Ruiz, Tzanina Saveta, Ondrej Zamazal, Sven Hertling, Michael Roder, Irini Fundulaki, Axel Ngonga Ngomo, Mohamed Sherif, Amina Annane, Zohra Bellahsene, et al. 2018. Introducing the HOBBIT platform into the ontology alignment evaluation campaign. In *13th International Workshop on Ontology Matching (OM)*, Vol. 2288. 49–60.

[32] Milos Jovanovik. 2017. *HOBBIT Examples source code on Github.* https://github.com/hobbit-project/SpatialBenchmark

[33] Milos Jovanovik, Timo Homburg, and Mirko Spasić. 2021. A GeoSPARQL compliance benchmark. *ISPRS International Journal of Geo-Information* 10, 7 (2021), 487.

[34] Nikolaos Karalis, Georgios M. Mandilaras, and Manolis Koubarakis. 2019. 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 (Lecture Notes in Computer Science, Vol. 11779)*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon (Eds.). Springer, 181–197. doi:10.1007/978-3-030-30796-7_12

[35] Charalampos Kostopoulos, Giannis Mouchakis, Antonis Troumpoukis, Nefeli Prokopaki-Kostopoulou, Angelos Charalambidis, and Stasinos Konstantopoulos. 2021. 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, 664–679.

[36] Manolis Koubarakis, Konstantina Bereta, Dimitris Bilidas, Konstantinos Giannousis, Theofilos Ioannidis, Despina-Athanasia Pantazi, George Stamoulis, Seif Haridi, Vladimir Vlassov, Lorenzo Bruzzone, et al. 2019. From copernicus big data to extreme earth analytics. *Open Proceedings* (2019), 690–693.

[37] Kostis Kyzirakos, Manos Karpathiotakis, and Manolis Koubarakis. 2012. 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, 295–311.

[38] Davy Landman, Alexander Serebrenik, and Jurgen Vinju. 2014. Empirical Analysis of the Relationship between CC and SLOC in a Large Corpus of Java Methods. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 221–230. doi:10.1109/ICSME.2014.44

[39] Rabbit Technologies Ltd. 2007. *RabbitMQ Broker web site.* https://www.rabbitmq.com/

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

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

[42] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. 2011. 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, 454–469.

[43] Thomas Mueller. 2005. *H2 Database) web site.* https://h2database.com/

[44] MvnRepository. 2024. *Maven Central Repository web site.* https://mvnrepository.com/repos/central

[45] Axel-Cyrille Ngonga Ngomo, Sören Auer, Jens Lehmann, and Amrapali Zaveri. 2014. 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* (2014), 1–99.

[46] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. 2007. A SLOC counting standard. In *Cocomo ii forum*, Vol. 2007. Citeseer, 1–16.

[47] Ontotext. 2024. *GraphDB.* https://graphdb.ontotext.com/

[48] Ontotext. 2024. *GraphDB GeoSPARQL.* https://graphdb.ontotext.com/documentation/10.7/geosparql-support.html#geosparql-support

[49] Ontotext. 2024. *GraphDB ImportRDF GeoSPARQL.* https://graphdb.ontotext.com/documentation/10.8/command-line-tools.html#importrdf

[50] OpenLink. 2024. *OpenLink Virtuoso Performance Tuning.* https://vos.openlinksw.com/owiki/wiki/VOS/VirtRDFPerformanceTuning

[51] Oracle. 2019. *Spatial And Graph web site.* https://docs.oracle.com/en/database/oracle/oracle-database/19/spatial-and-graph.html

[52] Taha Osman and Gregory Albiston. 2022. GeoSPARQL-Jena: Implementation and Benchmarking of a GeoSPARQL Graphstore. In *23rd European Conference on Knowledge Management Vol 2*. Academic Conferences and publishing limited.

[53] Alisdair Owens, Andy Seaborne, Nick Gibbins, et al. 2008. Clustered TDB: a clustered triple store for Jena. (2008).

[54] Kostas Patroumpas, Giorgos Giannopoulos, and Spiros Athanasiou. 2014. 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*. 301–310.

[55] Matthew Perry, Ana Estrada, Souripriya Das, and Jayanta Banerjee. 2015. Developing GeoSPARQL Applications with Oracle Spatial and Graph.. In *SSN-TC/OrdRing@ ISWC*. 57–61.

[56] RDF4J. 2024. *RDF4J.* https://rdf4j.org/

[57] RDF4J. 2024. *RDF4J Lucene GeoSPARQL.* https://rdf4j.org/documentation/programming/geosparql/

[58] RDF4J. 2024. *RDF4J NativeStore API Doc.* https://rdf4j.org/javadoc/4.3.16/org/eclipse/rdf4j/sail/nativerdf/NativeStore.html

[59] Michael Röder, Denis Kuchelev, and Axel-Cyrille Ngonga Ngomo. 2020. HOBBIT: A platform for benchmarking Big Linked Data. *Data Sci.* 3, 1 (2020), 15–35. doi:10.3233/ds-190021

[60] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. ACM. doi:10.1145/2815072.2815073

[61] Michael Röder. 2018. *HOBBIT Java SDK web site.* https://github.com/hobbit-project/java-sdk

[62] Michael Röder. 2024. *HOBBIT Examples source code on Github.* https://github.com/hobbit-project/hobbit.examples

[63] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. 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, 52–69.

[64] Stardog. 2024. *Stardog.* https://www.stardog.com/

[65] Harsh Thakkar. 2017. 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, 256–266.

[66] Antonis Troumpoukis, Stasinos Konstantopoulos, Giannis Mouchakis, Nefeli Prokopaki-Kostopoulou, Claudia Paris, Lorenzo Bruzzone, Despina-Athanasia Pantazi, and Manolis Koubarakis. 2020. GeoFedBench: A Benchmark for Federated GeoSPARQL Query Processors.. In *ISWC (Demos/Industry)*. 228–232.

[67] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. 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*, Peter A. Boncz and Josep Lluís Larriba-Pey (Eds.). ACM, 7. doi:10.1145/2960414.2960421

[68] W3C. 2013. *SPARQL 1.1 Query Language web site.* W3C. https://www.w3.org/TR/sparql11-query/

[69] Wikipedia. 2008. *Geohash wiki page.* https://en.wikipedia.org/wiki/Geohash