GeoSPARQL and SPARQL Benchmarking with GeoRDFBench Framework

Theofilos Ioannidis
Dpt. of Informatics and Telecommunications, National and Kapodistrian University of Athens, Greece

Nikos Mamoulis
Dpt. of Computer Science and Engineering, University of Ioannina, Greece

Manolis Koubarakis
Dpt. of Informatics and Telecommunications, National and Kapodistrian University of 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. Furthermore, these JSON specification libraries can be served remotely by a configurable REST API endpoint. 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.

2012 ACM Subject Classification Information systems → Spatial-temporal systems; Information systems → Semantic web description languages

Keywords and phrases geospatial, semantic, benchmarking, framework

Digital Object Identifier 10.4230/TGDK.4.2.8

Category Resource

Supplementary Material

Software (GeoRDFBench Framework, code): `https://github.com/tioannid/geordfbench` [31]
archived at `swh:1:dir:972135253c56ac42c3face1b01ab2ad360acea96`
Service (GeoRDFBench Framework, web site): `https://geordfbench.di.uoa.gr/` [32]
Other (Paper Benchmark Results, Github): `https://github.com/tioannid/geordfbench_results` [36]
archived at `swh:1:dir:cbc8f90f62d28e3d3136adc697cbe936dec01bf1`
Other (Paper Benchmark Results, Zenodo): `https://doi.org/10.5281/zenodo.15349539` [35]
Software (GeoRDFBench Samples, code): `https://github.com/tioannid/geordfbench_samples` [33]
archived at `swh:1:dir:57f7a454372a393491bfed1715859b690acb5eee`
Dataset (Geographica 2, dataset): `https://doi.org/10.5281/zenodo.13283414` [34]

Received 2025-07-02 **Accepted** 2026-06-22 **Published** 2026-09-03

Editors Stefania Dumbrava, George Fletcher, Matteo Lissandrini, and Riccardo Tommasini

Special Issue Data Management for (Knowledge) Graphs

Generative AI Declaration No form of AI, generative or otherwise, has been used in any phase or step of design/development/testing/editing/formatting/etc., neither for the software presented in the paper not for the paper itself.

1 Introduction

Many projects [12, 15, 16] 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 [49]. The number of graph database systems with varying degrees of spatial support have steadily increased for the last years [21, 46, 9, 7, 64, 37, 43, 19, 39, 2, 5] and their scalability has been tested for managing very large LOD datasets such as in the DBLP knowledge graph and respective SPARQL endpoint [1]. 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 [27, 10, 50, 25, 20] have assisted in creating datasets of desired size and attributes, while log-mining techniques [50, 74, 20] 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 [70]. 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 [37, 9, 25] has led us to believe that the geospatial semantic store research area would greatly benefit from 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.

Our work, at this stage, focuses on benchmarking single node GeoSPARQL/SPARQL query engines through the console. It makes it easy to integrate 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 [70] 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 [37] benchmark's components, a PostgreSQL and an H2 [51] embedded implementation of the JDBC report sink and hosts with Linux and Windows operating systems*. The REST API endpoint module promotes a more distributed approach to the framework's operation, as JSON specifications can also be served remotely as URL calls.
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 [67] with or without Lucene SAIL [68], (ii) GraphDB [57], (iii) Stardog [75], (iv) Strabon [46], (v) Virtuoso [21], (vi) Jena GeoSPARQL [61]*. 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.

[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.

5. The framework can also be used for *SPARQL benchmarks*. LUBM [27] 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. A GeoSPARQL/SPARQL benchmarking framework's tasks and comparison criteria are detailed in Section 3. Section 4 presents the high level architecture of the framework while Section 5 explains GeoRDFBench's runtime. Showcasing the JSON generator API follows in Section 6. Section 7 shows an evaluation of the framework and Section 8 compares GeoRDFBench with LITMUS, IGUANA benchmarking frameworks, the HOBBIT platform and the Geographica 2 benchmark. Finally, Section 9 presents conclusions and future work. Complementary and supporting information with code listings, benchmark logs and the experiment execution algorithm is provided in a list of appendices.

2 Related Work and Background

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

2.1 Graph Store Categories

In general, Graph stores 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 [80] and a standardized spatial vocabulary GeoSPARQL [65]. LPG stores, on the other hand, excel in graph traversal and path search for analytics and machine learning. But, until recently, they lacked standardization as there are several data models and languages from high-profile vendors and institutions, such as Neo4j's Cypher [24], Apache TinkerPop Gremlin [71], the Oracle supported PGQL [79] and G-Core [3] from the Linked Data Benchmark Council (LDBC). As of April 2024, the first edition of the Graph Query Language (GQL) standard [38] is officially published and hopefully will allow language inter-operation with SPARQL, yet no standard spatial vocabulary has been proposed. Another issue is that some of these languages are declarative while others are procedural basically requiring expert knowledge specific for each system to efficiently query stored data. Neo4j's proprietary procedural Cypher language provides native support only for POINT geometries [4] and only through the external plugin Neo4j Spatial [53] support is extended to other geometries. This plugin aligns closely with the concepts found in spatial database systems, such as the layer concept found in Shapefile and Open Street Map geographic data file formats and efficiently querying data require proficient and explicit procedural manipulation of all relevant functionality.

The interoperability of RDF and Property Graphs has been studied in [4] and the G2GML mapping language between the two models has been proposed in [13]. The RDF community on the other hand, also realizing that triples' annotation with metadata, such as edge properties, can lead to data size blow-up and unnecessarily complex queries, defined the RDF*[28] data model and the SPARQL* language, which implement this sought out LPG feature and promote interoperability in the graph community. Neo4j's neosemantic (n10s) [54] toolkit offers support for RDF data and its related vocabularies, but unfortunately not the GeoSPARQL vocabulary.

To conclude, at the time of writing of this paper, benchmarking geospatial graph databases with complex geometries support, in a declarative manner, is limited to the RDF model.

[4] `https://neo4j.com/docs/cypher-manual/current/values-and-types/spatial/`

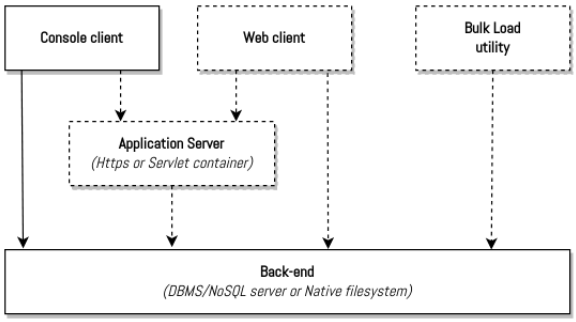


Figure 1 Geospatial Graph Store Architectures.

2.2 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 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 [7] uses the embedded key-value database Berkeley DB with a standard R-tree spatial index, Strabon [46] uses PostgreSQL and PostGIS with an R-tree over GiST [29] 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 [60] 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 sources.

2.3 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 [45, 9]. 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 [23] and geohash [81] 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[5], 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.

2.4 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 [27], BSBM [10], DBpedia SPARQL benchmark (DBPSB) [50] and the Social Network Benchmark (SNB) [20], just to mention a few. GeoSPARQL benchmarks have been presented in [63, 61], the benchmark Geographica in [25], a smart city services related benchmark in [8], a compliance benchmark in [42] and Geographica 2 in [37].

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.

2.5 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. For example, LUBM, which focuses on reasoning, features a university ontology-based synthetic data generator able to scale to arbitrary sizes. DBPSB's

[5] Removing problematic queries or reconfiguring the system.

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 [74] 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 [17] 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 [77] proposes uniform benchmarking of non-spatial data management systems supporting different query languages, such as SPARQL and Gremlin. Sparqloscope [6] is a dataset statistics-driven, parametric templates-based SPARQL query generator which creates a total of 105 queries for any given RDF dataset, covers a large subset of SPARQL 1.1 features and allows a feature at a time study. 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 [44] cloud benchmarking engine for federated query processors includes the GeoFedBench [78] benchmark, which focuses on validation of the actual crop land usage against the Austrian land survey dataset.

2.6 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 [70], the most complete of these platforms, extends the scope of benchmarking to the entire LD life-cycle [55], such as *link discovery* [40] employs intuitive web UIs and allows the integration of systems in various programming languages. For a comparison between GeoRDFBench and HOBBIT please refer to Section 8.

3 Comparison Criteria for GeoSPARQL Benchmarking Frameworks

In order to streamline the comparison of GeoSPARQL and/or SPARQL benchmarking solutions, a number of criteria categories need to be considered. Each of them comprises several criteria, however, we discuss the absolute necessary. GeoRDFBench's design aims at covering as many of these criteria as possible or at the very least be easily extensible to include additional ones.

3.1 Benchmarking Coverage

This discussion concerns benchmarking four (4) stages of working with RDF stores:

1. **RDF Data Ingestion (Loading and Indexing)**. This stage comprises measuring how long it takes for an RDF store to load an RDF dataset to a new repository (*load time*) and how long it needs to index the loaded data (*indexing time*). The *ingestion time* is the summation of load and indexing times. When multiple ingestion methods are available (online, offline/bulk loaders) the most appropriate method for the input RDF dataset size should be chosen unless comparing all loaders is an explicit aim of the benchmark. The stage completes by measuring the repository size after loading (*load size*) and after indexing (*ingestion size*). The repository *index size* is the difference of ingestion size minus the load size. We need to note that several systems have loaders that report only the repository ingestion time and size and might only offer an opportunity to take statistics about secondary indexes, such as spatial ones.
2. **Query Execution Performance and Accuracy**. This stage focuses on measuring the total time (*total execution time*) a query takes in order to execute (*evaluation time*) and return the entire resultset (*scan time*). Three important execution variants need to be considered

though: (i) single query with all caches cleared (*cold cache*), (ii) single query with caches enabled by previous executions (*warm cache*) and (iii) cache indifferent execution of constant or randomized mixtures of queries (*continuous*/batch/queryset/query mixes). For deterministic queries the gold resultset or at least the expected number of results is known, therefore it is also necessary to measure if the query returned the correct number of results (*query accuracy*). This test is necessary because small deviations in the number of results can reveal problems with function implementation or rounding errors in the internal representation of information on data files and indexes.

3. **RDF Store Data Scalability and User Concurrency**. Nowadays, behavior predictability against unknown future loads (data size, number of users) carries important weight on organization decision making process about the most appropriate RDF solution for their needs. As such, the study of the behavior of RDF stores against increasing larger datasets (*scalability*) and number of users (*user concurrency*), which actually takes part in the previous phase, is presented as a separate phase.
   Common knowledge dictates, that excluding embedded system databases, serving concurrent users is rarely a need or a problem if the underlying repository size is small to medium but becomes an issue when data size drastically increases. This is also verified in the conclusions of benchmarks performed with the IGUANA framework involving several RDF stores against real-world and synthetic workloads: "..while the triple stores we evaluated seem to scale up well to concurrent queries and updates, *they are all affected significantly by the size of the datasets*". Therefore, we propose that a logical sequence is testing system scalability first and later on perform user concurrency testing in the scalable systems subgroup.
4. **Statistics Reporting and Logging**. In this final stage the aim is to generate various useful statistic measures (median, average, standard deviation, no of queries per hour, etc.) about each query, queryset or system that might help the user draw conclusions and/or shed light on problematic areas that need to be addressed with further investigation. The recorded information of the first three stages augmented with the statistics produced in this phase can then be organized and disseminated in the form of reports in various locations (file system, databases) and different formats (CSV, TSV, JSON, XML, RDF, tables). Based on these reports, graphs and charts might be generated according to the users preferences. Also, in dynamic or template based querysets it is very important to log the exact query instance that was executed and for which metrics were collected. Finally, system configurations and experiment host environment parameters need to be recorded as a form of "chain of evidence" when comparing systems on the same host or the same system in different hosts.

3.2 Geospatial Support

RDF stores can have full or partial support of GeoSPARQL or even alternative spatial vocabularies with equivalent function libraries. The evaluation of RDF stores' spatial support is tested by the existence of the following points:

1. **Spatial Index Creation Time and Size**. As mentioned earlier, when possible, the time it takes to create the spatial index (*spatial indexing time*) and the additional size imposed by it to the repository (*spatial index size*) should be measured. In most cases, these two values are substantial compared to the repository load time and size. They represent an additional investment of valuable resources and should carry a weight factor in the final evaluation of whether the query execution benefits, they might offer, are worth it at all or if in some application use cases they are.
2. **Spatial Index Specification**. Each RDF store specifies a list (usually one only) of *spatial indexing algorithms* (quad, geohash, b-tree, etc.). Some of these algorithms have accuracy ranges and a default value (*accuracy value*) which dictates how detailed a spatial search in

different regions will be. Low accuracy values have a triple side-effect: (i) they might allow false-positive results to appear for a specific spatial filter, (ii) the search time is reduced and (iii) the spatial indexing time and index size are also reduced. Thus it affects the measured query accuracy, the total execution time, the ingestion time and size. It is therefore important, when systems' configurations allow it, to set all competing RDF stores' spatial searching capabilities as close as possible. Some systems though, such as, RDF4J, Stardog, etc., they do not offer a way to set the accuracy value, while others such as GraphDB offer dynamic reconfiguration of the spatial index.

3. **GeoSPARQL Support and Compliance**. GeoSPARQL provides a common vocabulary and language for spatial RDF stores to be tested side-by-side either for performance or for feature coverage (*compliance level*) measurements. When possible, GeoSPARQL querysets should be used to drive the relevant benchmarks on spatial triple stores. However, there were and still are systems that have partially implemented GeoSPARQL and missing functionality is offered by system-oriented spatial vocabularies[6] and library extensions of SPARQL. For such systems, GeoSPARQL querysets might need some level of translation to system-specific variants (*query translation*). Past benchmarking efforts actually followed this path in order to be more inclusive. We believe that benchmarking frameworks should cater for such translations (application hooks) from GeoSPARQL constructs to system-oriented SPARQL constructs making certain that they are applied consistently generating variants of the input GeoSPARQL querysets on the fly while reporting the query variant executed in each case.
4. **GeoSPARQL Spatial Indexing Challenges**. All GeoSPARQL engines do not automatically recognize the presence of `geo:wktLiteral` property in order to spatial index the WKT serializations of the geometry objects referred to. They may require that each dataset explicitly states the properties that point to the geometry serializations e.g., the RDF4J Lucene SAIL[7] requires that to properly work. Some other systems, such as Parliament may require inference be enabled and additional RDFS triple statements inserted to point to the sub-properties of GeoSPARQL's `geo:hasGeometry` property. A benchmarking framework should enable the user to configure these properties in the most convenient place, preferably in a declarative manner.
5. **Spatial Index Enabled GeoSPARQL Query Execution**. Even during query execution some systems like Parliament and GraphDB will enable their spatial index only when GeoSPARQL spatial predicates are used. In these cases benchmarks such as Geographica and Geographica 2 do offer equivalent querysets to make sure that these systems are given fair opportunity at showing their strengths. Benchmarking frameworks should offer querysets with bundled spatial function and predicate versions of their queries and make sure that the execution engine feeds the appropriate set to the system that requires it. For systems such as GraphDB which supports GeoSPARQL operations with and without the use of the spatial index the engine should allow the system to choose which one of the two options to use.

3.3 Semantic Web Standards Support

Apart from the specific GeoSPARQL support mentioned, benchmarks and relevant frameworks should also check if several other Semantic Web standards are supported by the RDF stores under test. We briefly mention the most important ones:

[6] Virtuoso supports virtrdf:Geometry literal and built-in bif:xx functions
[7] `https://rdf4j.org/javadoc/latest/org/eclipse/rdf4j//sail/lucene/LuceneSail.html#WKT_FIELDS`

1. **Quads as well as Triples**. Many benchmarks and frameworks do not even mention or test if triple stores support quads. They focus entirely on triples and they are missing the evaluation and study of the relevant 4-level thematic indexing structures, e.g., RDF4J's `"SPOC, POSC, COSP"`[8] or Virtuoso's `"OGPS, PGOS"`[9].
2. **Multiple RDF Serializations Support**. It follows from the above that *TriG*, the extension of Turtle serialization, which supports representing a complete RDF Dataset, is not tested by all benchmarks or given as an input option in benchmarking frameworks. The most common format for datasets is *N-Triples* which uses absolute IRIs and is the least space conserving one.
3. **SPARQL Query Namespace Prefixes**. Some benchmarking frameworks, in an attempt to reduce technical effort, narrow their focus and do not provide any assistance in building the appropriate SPARQL query namespace prefix header for each executing system. Queries with their header are commonly managed and consumed as a complete block either one query per file or line. However, if query translation is required for some systems, this effort becomes more difficult to implement (usually manually) and verify its outcome. The solution they propose is a different set of query files or folders for each system, which the user has to consider when building the experiment tasks or workloads. The actual namespace prefix header for each query is a merge of the specific query's prefix map, combined with the implied prefix map in the target dataset or some graphs in it and finally the system-oriented vocabulary and extension prefix map (*namespace prefix map merging*). An automated namespace merging implies separate handling of the query from its header which by design simplifies the query translation process. One should expect from benchmarking frameworks claiming to assist users to provide some short of automated support for this sort of tasks.

3.4 Framework Automation Level

The most important feature of a benchmarking framework over specific benchmarks is the amount of assistance it provides to its users.

1. **Setting Up New Benchmarks**. Designing a new benchmark usually means selecting and crafting datasets, querysets and describing the execution logic of the experiments. Usually a researcher locates the real data sources and through data curation and appropriate conversions formulates the desired RDF real-world datasets which may be structured as a collection of RDF graphs. Querysets can be a collection of static queries that are designed to test specific aspects of the systems against the chosen datasets. If the collection comprises many queries which follow certain patterns then *template based queries* are used where static or dynamic replacement of template tokens occurs to create the *ground queries* and at the same time keeps the number of queryset entries very low. More dynamic scenarios might include *synthetic query generators* which follow some ontology-based logic. In these cases they might be paired with a *scalable synthetic data generator* based on the same ontology which can generate datasets of various sizes. Yet another very popular case is to use log mining techniques from application logs and through filtering and transformation process generate the final queryset. This technique however targets real-world datasets and specific application use cases. Benchmarking frameworks therefore should at the very least offer: (i) a scalable synthetic data and query generator, (ii) facilitate dataset integration and bundling, (iii) support for a query template mechanism that allows easy creation of queryset variants while keeping the size small and (iv) support the three query execution variants mentioned in Section 3.1.2.

[8] (S)ubject, (P)redicate,(O)bject,(C)ontext
[9] (P)redicate,(G)raph,(O)bject,(S)ubject

2. **Updating Existing Benchmarks**. Updating an existing benchmark usually implies refreshing all participating systems to newer versions and probably adding new systems. The researcher faces an increased number of features e.g., new indexing algorithms, most of the time improvement of capabilities, a need for upgrading supporting client RDF library versions, changes in back-end architecture e.g., addition of an application or database server component, a new array of optimal system configuration options. A benchmarking framework should encode, as much of this information as possible for each supported system, in scripts, property files or using dynamic discovery in order to guide the user. It should also be host resource aware and allocate reasonable resources following the proposed performance guidelines of each system. This action assists the user in system configuration on the host environment which is a very laborious and sensitive task. Sub-optimal configuration of a system may result in biased results and wrong conclusions in the comparative evaluation report.
3. **Repeatable and Adaptable Experiment Execution**. With time, not only RDF stores evolve but also available host hardware and operating systems improve. Since benchmarking is a process that must be repeated to reflect the current status of the competition, it is necessary that benchmarking frameworks facilitate running the same benchmark in different environments. This can only be achieved by isolating the host and operating system aspects (*benchmarking environment*) that are relevant to the process, model them and create declarative specifications for each prospective experiment environment. This should ideally provide a plug and play behavior which the framework's executing engine should be able to adapt to. Aspects that are commonly used and affect the benchmarking process are host available RAM, disk and file locations for datasets and repositories, operating system cache control and console commands, etc. Finally a common scenario that benefits from experiment execution adaptability is when there is a need for running queryset subsets. The case might be that the experiments must focus on a specific area of interest, e.g. spatial joins with intersect operator, or that the target RDF stores under test have limited capabilities and cannot execute the full queryset. *Queryset filtering* for selective query execution at experiment run time promotes clarity and allows for maintaining well designed queryset libraries without having to deal with a haphazard collection of multiple variants of the same base querysets.
4. **Native and Dockerized Execution**. Until now, the previously mentioned problem had been addressed through the use of one or more containers (*dockerized execution*) which however use the same host operating system and only modify the allocated resources. Few frameworks allow *native execution* and testing of the systems and even fewer allow both types of operation. The problems of dockerization can be summed to the choice of using the `privileged mode` of operation or not. If not used, the framework is unable to perform cold cache query executions and if it does, it breaks the host-container isolation[10] creating security risks and instability in the host and other running containers. LITMUS is a framework that uses a single privileged container approach, IGUANA and GeoRDFBench use native mode of execution and the latter also allows single non privileged container operation for the initial benchmarking phases or training purposes.

3.5 Sustainability and Extensibility

We conclude with two obvious but neglected aspects of a benchmarking framework. The first question is whether this resource is actively supported, that is if there new features added or bugs fixed in the past 12-24 month period. Is there a plan for the medium and long-term maintenance of

[10] `https://docs.docker.com/reference/cli/docker/container/run/#privileged`

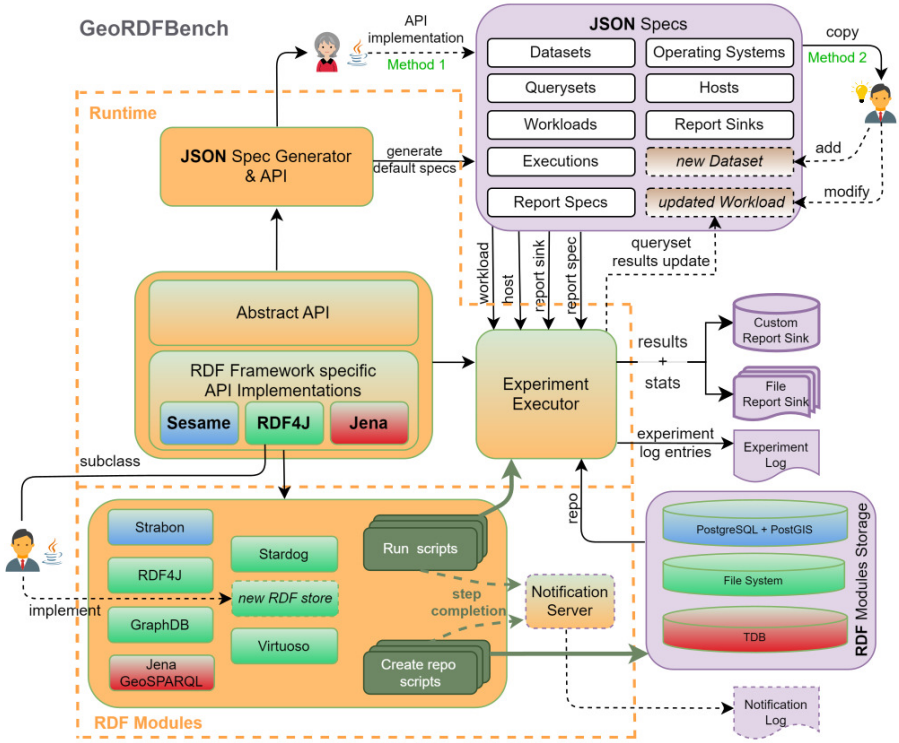


Figure 2 Architectural overview of GeoRDFBench Framework.

the resource (*sustainability*)? The next question is about the extensibility of the resource. Can it be easily extended or adapted by other users to support research and experimentation? Extending a framework which is *actively supported* is easier and can be more productive because questions can be readily answered by the developers. Another important factor is whether the technologies and methodologies in design and implementation are *open source, good quality and widely used*. The final and most crucial factor is of course if the resource was intentionally *designed to promote easy continuous evolution and extension*. Monolithic script-based approaches can rarely cover the last point as they require complete rewrite for changes that pertain to the execution logic, the introduction of template based query support, handling of system-depended vocabularies and query rewrites.

In the rest of this paper the above criteria are referenced by number in square brackets after the relevant text.

4 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*

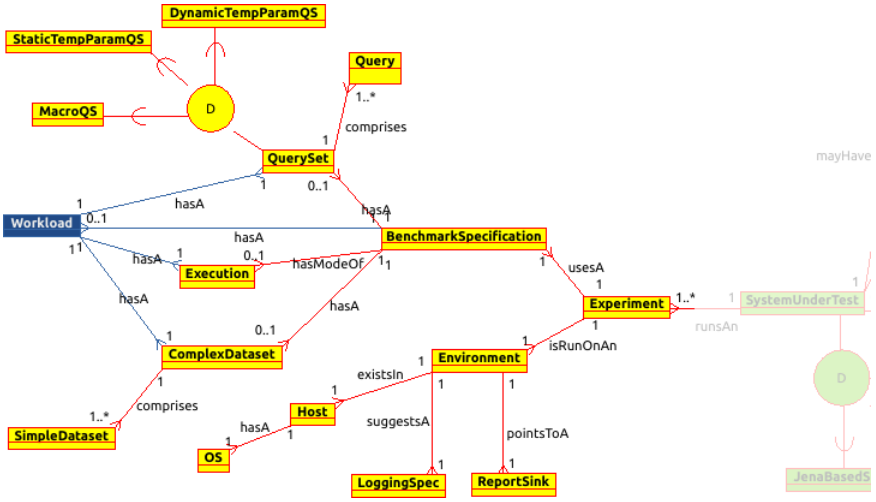


Figure 3 EER diagram of Abstract API - Experiment Components.

to create repositories and import data to them. Each store's *experiment run script* initiates a benchmark experiment and eventually invokes the runtime's *experiment executor* component passing all required inputs which include, among others: the RDF store, the repository, the benchmark workload, the host where the experiment is conducted on and the report sink where experiment results and statistics will be stored. Both types of scripts send progress messages to the optionally enabled, remote or local, *notification server* which logs them and serves as a useful non-intrusive monitoring tool for the researcher.

The default *JSON specifications* correspond to the Geographica 2 and LUBM benchmarks' components and are generated by the *JSON Specs Generator*. This "starter dough" library is stored on the file system separately from GeoRDFBench code. The user can either implement new custom specifications using the JSON Specs Generator API (**Method 1**) or copy and modify existing ones (**Method 2**).

The *JSON Library REST API endpoint* module (not shown in Figure 2) can provide an alternate way for the experiment executor to acquire JSON specifications from a remote location such as a docker container or server and promotes both a more distributed approach to the framework's operation and increased reusability of these resources.

5 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.

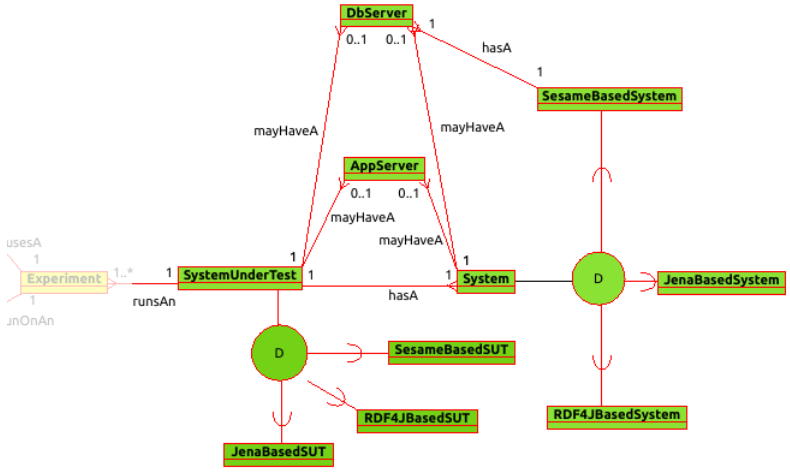


Figure 4 EER diagram of Abstract API - System Under Test.

5.1 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[11] 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.

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.

5.1.1 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.

5.1.1.1 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 [3.4.1]. 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

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

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` [3.2.3, 3.2.4], 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) [3.3.1, 3.4.1] and the list of simple datasets comprising it. Experiments use only complex datasets.

Queryset

GeoRDFBench follows the path of Geographica 2 and adopts the **micro** and **macro** experiment queryset categorization proposed in the Jackpine [66] geospatial database benchmark. Micro querysets simulate a lab-like controlled environment where the aim is to assess the performance and compliance of individual functions of interest e.g., a spatial function, and each query has a unique and autonomous purpose. The performance metrics of interest are query accuracy and the query response time over repeated executions with cold and warm caches [3.1.2]. Macro querysets, on the other hand, attempt to simulate real-world scenarios where the aim is to assess the expected performance of certain use cases e.g., "Rapid Mapping for Wild Fire Monitoring" in Geographica 2. They can comprise a random selection of *ground queries*[12] with arbitrary criteria or even form a dependency graph of ground and *template queries*[13] where the resultsets of specific ground queries' execution operate as arguments for template parameters in order to instantiate the template queries before they execute [3.4.1]. The performance metrics of interest in these cases is mainly query throughput, irrespective of cache status, which translates to number of query executions in a period of time e.g., 1 hour [3.1.2].

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 [3.2.3, 3.2.5] 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 [3.1.2]. The *queryset* specification contains: a logical name, the queryset path location, a map of queryset relevant namespace prefixes [3.3.3], 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* [3.4.3]. All QuerySet subclasses can handle ground queries but they differ in the flexibility they offer in template query parameter instantiation. The `StaticTempParamQS` subclass models sets of template parameter queries which have fixed parameter values for all queries. This is useful when very large literals e.g., a 2KB WKT polygon literal, need to be repeated in several queries, thus it allows for a succinct representation of the template queryset for viewing, storing and dissemination purposes while allowing for the exploded representation only during experiment execution [3.4.1]. The `DynamicTempParamQS` subclass models sets of template parameter queries which may have different value for each parameter for each query. This can be viewed as providing a declarative alternative for creating a simplified synthetic query generator [3.4.1]. While the previous subclasses are useful for micro experiment types, the `MacroQS` subclass and its specializations[14] are useful for macro experiment types.

[12] Queries with RDF terms only (IRIs, blank nodes and literals).
[13] Queries with replacable tokens acting as placeholders for runtime substitution.
[14] Not shown in Figure 3 for simplicity reasons.

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 in the standard file-system based statistics and the policy to follow when a cold execution times out [3.1.2, 3.4.3, 3.5]. The non-default *print* action triggers a pseudo-execution which generates the ground queryset for inspection purposes [3.1.4].

Workload (compact form)

The entity `Workload`, in Figure 3, represents the compact form of the benchmark experiment description: *dataset + queryset + execution specification.* It promotes ease of use and consistent experiment execution [3.4.1-3].

5.1.1.2 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) [3.4.2, 3.4.3].

Host

The entity `Host`, in Figure 3, represents the hardware platform where the benchmark experiment is taking place [3.4.2, 3.4.3]. It has the host name, the operating system, IP address, total RAM (GBs), the base path for the actual dataset files, the base path for RDF store repository files and the base path for the standard file-system based reports and statistics.

Report sink

In addition to the standard file-system based reports and statistics that GeoRDFBench generates by default, there is the option of collecting these statistics in an advanced report store such as a database [3.1.4]. The entity `ReportSink`, in Figure 3, describes the experiment result report store, where detailed statistics will be sent and custom reports generated. 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's database schema is automatically generated by the runtime using the *database definition script* bundled with each report sink's source code. This schema can also be generated with the help of the runtime-bundled *database generation SQL script.*

Customized reports are provided by several report views with aggregate statistics which exist in the database schema. These include the mean and median value of the total query execution time, the number of executions per query, cache and system tested. Additional statistics such as variance and standard deviation can be easily added to these report views either by altering the report view definition with the database client or more permanently by updating the `schemaInitScriptreport` static property of the relevant abstract class `JDBCRepSrc` or `EmbeddedJDBCRepSrc` in the GeoRDFBench's runtime.

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 [3.1.4]. 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.

5.1.2 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 [3.4.3, 3.5]. 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 a *System* with its optional application and database server components, represented by *AppServer* and *DbServer* respectively [3.4.3, 3.5].

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.

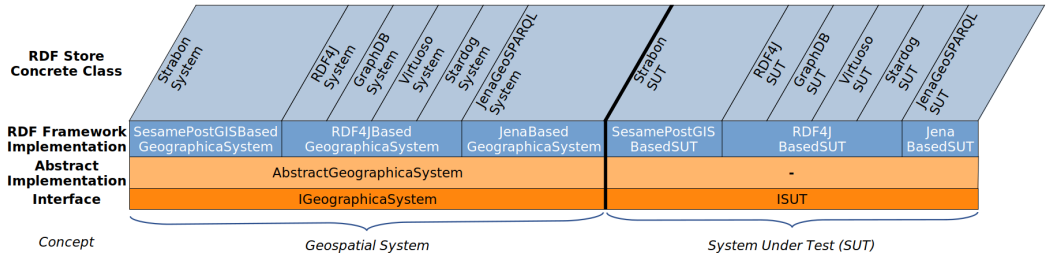


Figure 5 Systems & SUT Class Hierarchies.

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 [3.3.3, 3.4.2, 3.5].

5.2 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* [3.4.2, 3.5].

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.

5.2.1 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`

Jena API (`JenaBasedGeographicaSystem`)

Since only Jena GeoSPARQL uses this API, we simply chose a recent and frequently used Jena version in Maven Central [52] , from the latest stable branch, which was 4.10.0. Jena Tuple Database (TDB2) [62] 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`

5.2.2 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 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 [3.3.3, 3.4.2, 3.5].

5.3 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, whose logic is described in Algorithm 1 of Appendix C [3.4.2, 3.4.3, 3.5].

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 [3.3.3, 3.4.2, 3.4.3].

5.4 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 [3.2.3, 3.3, 3.4.1, 3.4.2, 3.4.3].

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 [22] JSON library is used to annotate interfaces and class hierarchies to simplify serialization and deserialization [3.5].

6 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 [3.1.2, 3.1.3].

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[15] 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 13 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		

6.1 Dataset generation

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

Listing 1 Generate Dataset Specification.

```
public static GeographicaDS newScalabilityDS() {
  // create a simple dataset object with a single N-Triples file
  GenericGeospatialSimpleDS sds
    = new GenericGeospatialSimpleDS("scalability_10K", // simple dataset name
      "Scalability/10K", // relative directory where the file resizes
      "scalability_10K.nt", NTRIPLES_STR); // the triples file
  // add to it any namespace prefixes used in the dataset file
  sds.addUsefulNamespacePrefix("lgo", "<http://data.linkedeodata.eu/ontology#>");
  // add to it any property used in the dataset that denotes feature geometry
  sds.addHasGeometry("scalabilityHasGeometry", "<http://www.opengis.net/ont/geosparql#hasGeometry>");
  // add to it any property used in the dataset that denotes WKT serialization
  sds.addAsWKT("scalabilityAsWKT", "<http://www.opengis.net/ont/geosparql#asWKT>");
  // create a complex dataset object with a single simple dataset object
  GeographicaDS gds =
```

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

```
      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 and Listing 11 details the generated JSON file.

6.2 Queryset generation

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

Listing 2 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);
}
```

The generated JSON file detailed in Listing 12 of Appendix A uses the `StaticTempParamQS` subclass which allows the user to model querysets with or without template parameters but with fixed parameter values for all queries.

6.3 Execution spec generation

The following code generates the Scalability execution specification object.

Listing 3 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 generated serialized representation is detailed in Listing 13 and fully describes the experiment execution model in *Scalability Workload Description* presented earlier in this section.

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* [33] application that is using as test case, the LUBM benchmark for SPARQL.

7 Experimental Evaluation

In this section, we present the process of running some of the Geographica 2 [37] 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.

7.1 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 [3.2.4]: (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` with *STRtree*[16] spatial index and (vii) Strabon `v3.3.3-SNAPSHOT`. Additionally, we tested the second GraphDB variant against two versions of the querysets [3.2.5]: (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[17] 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	Size	# of Features	# of Points	# of Lines	# of Polygons
10K	3.7 MB	1,135	587	0	900
100K	20 MB	12,166	6,623	4,239	2,531
1M	194 MB	118,161	46,781	45,238	29,200
10M	1.9 GB	1,038,739	317,865	328,630	427,842
100M	19 GB	10,259,959	904,677	2,058,386	7,553,440
500M	97 GB	48,623,878	5,520,767	15,771,932	23,390,220

7.2 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, 1M and 10M variants of the Scalability workload (see Table 1) where the respective datasets contain 10K, 100K, 1M and 10M triples [3.1.3]. 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* [3.2.3]. 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 are described in Table 2.

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

Listing 4 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#>
```

[16] From the JTS library, a query-only R-tree using the Sort-Tile-Recursive (STR) algorithm
[17] Stardog requires a license otherwise experiments fail.

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

For comparison, the next listing shows the spatial predicate queryset for GraphDB+P [3.2.5]:

Listing 5 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* [3.3.3]. 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 6 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 4 and 5 GeoRDFBench conveniently takes the necessary steps. First, the template parameters of the queries in Listing 12 are *replaced with their values during queryset deserialization*. After that, it *applies any system provided query*

translations in order to customizereate the ground queries [3.2.3, 3.4.1]. 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 7 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 13.

7.3 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 four scalability workloads (10K, 100K, 1M, 10M) there are `thirty two (32)` 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, 106	**RDF4J**	RDF4J	v4.3.15
4-6, 111	**JenaGeoSPARQL**	Jena GeoSPARQL	v4.10.0, *STRtree*
7-9, 108	**GraphDB**	GraphDB	v10.8.5
10,11,14, 121	**Virtuoso**	Virtuoso Open Server	v7.2.14, *2D R-tree*
15-17, 109	**GraphDB+**	GraphDB + *GeoSPARQL plugin*	v10.8.5, *Quad-11*
18-20, 110	**GraphDB+P**	GraphDB + *GeoSPARQL plugin*	v10.8.5, *Quad-11, Pred*
21-23, 107	**RDF4J+**	RDF4J + *Lucene Sail*	v4.3.15, *Lucene spatial index*
31-33, 124	**Strabon**	Strabon	3.3.3-SNAPSHOT

GeoRDFBench provides the following common tools [3.4.2, 3.5]: (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 the user 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 will probably want to comment out the repositories that are not required or add new repositories to them.

The repository generation wrapper script calls the **repository generation script** for each repository [3.2.1, 3.2.2, 3.4.2]. This script is highly non-uniform as it engulfs each system's approach on optimizing data loading, thematic and spatial index building. For each system the most performant available loader (default/online vs bulk/offline) was selected even if that was a console tool instead of a Java library API call and the maximum statistics provided by it was collected. The key takeaways we discovered are: (i) Virtuoso's loader, does not provide a spatial index creation time, (ii) Stardog creates spatial index only during repo creation and requires different configuration settings for loading and querying, (iii) GraphDB Preload requires the server to be offline and cannot work with inference, (iv) JenaGeoSPARQL does not report a spatial index size because it does not persist a spatial index on disk. For the above reasons repository generation was not handled by the runtime API but pushed down to the module level and handled through scripts. When possible (RDF4J, Strabon) repository creation has been implemented at the module level and therefore can be performed programmatically through GeoRDFBench [3.5].

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 8 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. Similarly, 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. The above experiment execution script differences can be found in Appendix B Listings 14,15.

7.4 Experiment log files

7.4.1 Environment variables log

There is one such log for each system configuration [3.1.4]. 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 9 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
```

7.4.2 Repository generation log

There is one such log for each system configuration [3.1.4]. 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. The RDF4JSUT+ output for the *scalability_10K* repository can be found in Appendix B Listing 16.

7.4.3 Run workload log

Each such log concerns a single run of a system against a workload. It has homogeneous output layout which comprises three sections [3.1.4]. In the first section all validated arguments are listed, the JSON specifications (which can be either JSON files or URL calls to the JSON Library REST API endpoint) 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. A sanitized version of the first section for Virtuoso's run against the Scalability 1M workload can be found in Appendix B Listing 17.

The second section details each query's execution steps. It begins with the first iteration of the first query SC1 with COLD caches. The ground query is displayed, system caches are cleared since it is a COLD cache execution and after a 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 25% of the total query timeout (24 hours)[18] specified in the execution specification. The executor reports all state transitions for the query execution and during the scanning phase reports the first 3 resultset entries according to the `simplereportspec_original.json` logging specification. Before exiting, the executor reports that it completed with no errors (COMPLETED-NONE), the query evaluation, scanning and total time in nano seconds, the number of results, the number of scan errors[19] and that the accuracy could not be verified, because the queryset specification provided did not include the expected number of results for this query. All query response times are net since they explicit aggregate only the query relevant portions of the code within the `run()` function of the executors' `Runnable` interface and exclude display, logging and when possible even exception handling logic. In Listing 10 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 10 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
```

[18] The Scalability query timeout is very high because of the very high query response times in the 500M dataset.
[19] 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 *geordfbench* 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 in Appendix B Listing 18.

7.5 Evaluation Results

Tables 4-6 present the evaluation results for the GeoRDFBench experiments [3.1.1-3, 3.2.1].

7.5.1 Repository size

Virtuoso is the most scalable store of all, when taking into account that its repos include a *2D R-tree* persisted spatial index. Its 10M repo is smaller than Jena GeoSPARQL's repo which does not persist a spatial index on disk and only 15% larger than the RDF4J's and GraphDB's repos which do not feature a spatial index at all. The fact that Virtuoso exhibits much larger repo sizes for small to medium datasets, points out that its internal database and index formats carry an almost constant overhead which is significant only in smaller databases and for more real case scenarios can safely be ignored. RDF4J's *Nativestore* [69] (B-Tree indexed, no spatial index) performs best in small to medium dataset sizes and very well for tens of millions of triples which is on a par with its specification. GraphDB's default configuration is storage efficient across all input dataset sizes. For small datasets it ranks average having a substantial overhead compared to RDF4J which however practically becomes irrelevant after the one million triples limit. In absolute terms it is the most scalable system for repository size, however we need to point out that similarly to RDF4J's Nativestore there is no spatial index which will incur additional search costs for high selectivity spatial queries. JenaGeoSPARQL's *TDB2* despite persisting 3 spatial indexes in memory and not on disk, seems to have a scalability issue. For smaller datasets it performs near the top of the competition, it then gradually degrades to an average score for datasets around one million triples and for bigger datasets it generates larger repositories even compared to the majority of systems with a persisted spatial index. By comparing Table 4 with Table 1 we also notice that the final repository sizes of JenaGeoSPARQL are almost identical to the input dataset sizes. In RDF4J+ the *Lucene* quad spatial index, increases the storage footprint by 45% on average with respect to RDF4J's Nativestore. GraphDB+'s *GeoSPARQL plugin* uses the same Lucene library as RDF4J+ for spatial indexing, also increases the storage footprint by 45% but scales better than it in a similar manner that GraphDB outperformed RDF4J. The underlying reason for this behaviour is that despite GraphDB being built on top of RDF4J it brings in an augmented design and optimizations that start to pay off for larger dataset sizes. Strabon's storage footprint is the largest of all system configuration due to the *PostGIS* database with an *R-tree over GiST* [29] spatial index.

7.5.2 Ingestion time

Table 5 presents the data ingestion times. When separate measurements for data loading and spatial indexing could be acquired they are provided. If spatial indexing was available but could not be measured the value was left empty while a hyphen (-) REPRESENTS stores that do not build a spatial index.

In absolute terms, RDF4J has the fastest ingestion times for small to medium dataset sizes and performs very well for tens of millions of triples. With the addition of Lucene spatial index RDF4J+ more than doubles the total ingestion time for smaller datasets and at best doubles it

Table 4 Scalability Workload Evaluation - Repository sizes (MB).

System ConfigName	Spatial Index	10K triples	100K triples	1M triples	10M triples
RDF4J	–	4	14	135	1262
JenaGeoSPARQL	memory	5	22	207	1977
GraphDB	–	28	38	144	1151
Virtuoso	disk	74	78	504	1338
GraphDB{+,+P}	disk	32	44	201	1637
RDF4J+	disk	7	21	205	1813
Strabon	disk	24	52	349	3078

for the largest one. JenaGeoSPARQL is second best for the first three datasets but its scalability potential starts to suffer at around ten million triples. For all GraphDB variants the data loading portion is almost identical and shows a time overhead for the first datasets. This is due to GraphDB's *Preload* [59] 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. This is confirmed in the largest dataset where GraphDB exhibits the best scalability. The Lucene based *GeoSPARQL plugin* in GraphDB+ variant more than doubles the total ingestion time. Strabon's three phase[20] 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 large time spent in phase one, RDF to CSV conversion, which is performed with the Redland raptor parser and serialization library. The average score of Virtuoso in the smaller datasets cannot overshadow the astonishing fact that it almost tied GraphDB's no-spatial index performance while building a spatial index. To put this into perspective, for the 10M triple dataset all other systems required more than 95 sec for building just the spatial index, while Virtuoso completed data ingestion and spatial indexing in approximately the same time.

Table 5 Scalability Workload Evaluation - Data ingestion (loading, spacial indexing) times (sec).

System ConfigName	10K triples			100K triples			1M triples			10M triples		
	Loading	*Indexing*	*Total*	*Loading*	*Indexing*	*Total*	*Loading*	*Indexing*	*Total*	*Loading*	*Indexing*	*Total*
RDF4J	0.62	–	0.62	1.80	–	1.80	11.14	–	11.14	105.38	–	105.38
JenaGeoSPARQL	2.49		2.49	3.91		3.91	16.61		16.61	121.90		121.90
GraphDB	14.89	–	14.89	11.82	–	11.82	17.59	–	17.59	97.68	–	97.68
Virtuoso	15.87		15.87	17.88		17.88	28.07		28.07	99.71		99.71
GraphDB{+,+P}	15.85	28.97	44.82	10.26	37.77	48.03	18.92	40.21	59.13	88.85	103.85	192.70
RDF4J+	0.62	1.98	2.60	1.80	2.62	4.41	11.14	12.49	23.63	105.38	94.29	199.66
Strabon	3.42	17.00	20.42	26,91	24.00	50.91	234.32	53	287.32	2504.14	371.00	2875.14

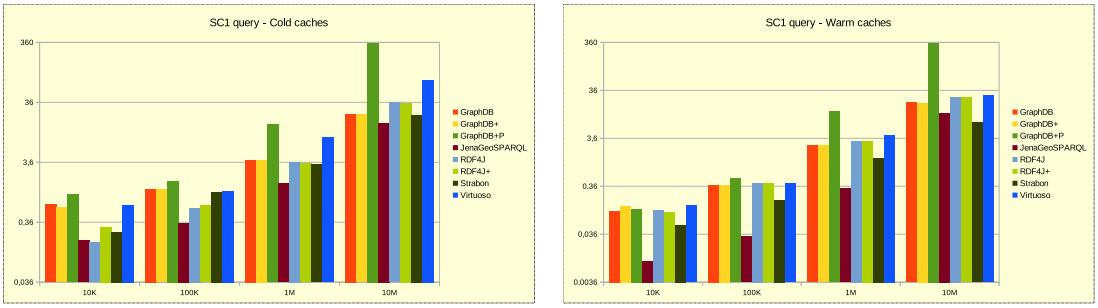
7.5.3 Accuracy & Response time

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. The response times are the median values out of three executions. Figures 6-8 show chart diagrams with the system response times for the three queries [3.1.4]. 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 R-tree spatial indexing process and/or in the implementation of `geof:sfIntersects` GeoSPARQL function which mishandles a border result.

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

Table 6 Scalability Workload Evaluation - Accuracy & Response Times.

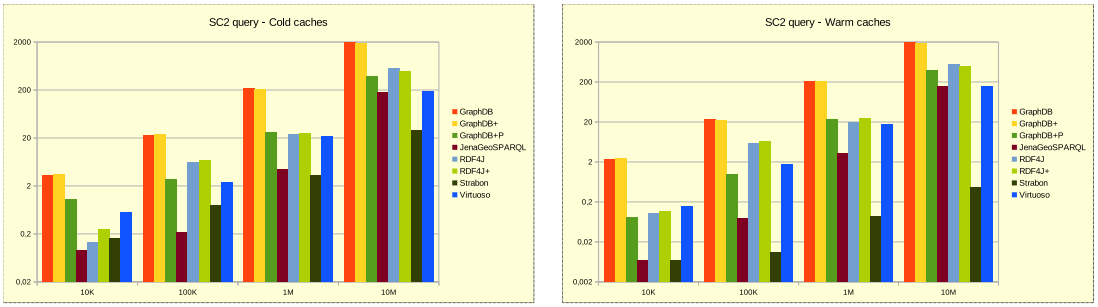
			RDF4J		JenaGeo-SPARQL		GraphDB		Virtuoso		GraphDB+		GraphDB+P		RDF4J+		Strabon	
Cache	Query	Workload	# Res	Time(s)	# Res	Time(s)	# Res	Time(s)	# Res	Time(s)	# Res	Time(s)	# Res	Time(s)	# Res	Time(s)	# Res	Time(s)
Cold	SC1	10K	554	0.167	554	0.179	554	0.710	554	0.703	554	0.634	554	1.036	554	0.298	554	0.246
		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
		10M	442580	35.818	442580	16.371	442580	23.008	442580	84.408	442580	22.549	442580	351.622	442580	35.028	442580	22.129
	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
		10M	9160	568.296	9160	176.511	9160	1957.782	9160	189.685	9160	1886.617	9160	388.638	9160	499.867	9160	28.72
	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
		10M	239	215.219	239	176	239	41.842	239	192.008	239	40.461	239	1937.18	239	135.291	239	49.696
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
		10M	442580	25.941	442580	12.25	442580	20.091	442580	28.146	442580	19.794	442580	350.648	442580	25.307	442580	7.692
	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
		10M	9160	555.843	9160	158.535	9160	1950.94	9160	151.939	9160	1896.749	9160	396.026	9160	484.631	9160	0.471
	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
		10M	239	199.875	239	169.945	239	41.285	239	151.545	239	40.247	239	1980.143	239	116.536	239	1.144



(a) (b)

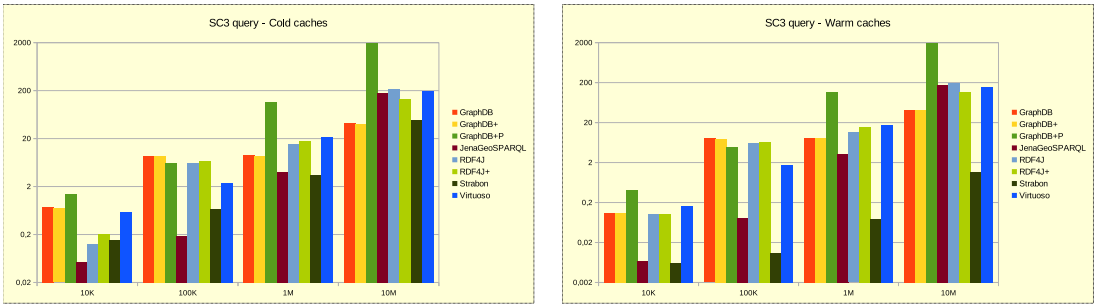
Figure 6 Response times – SC1 spatial selection query.

JenaGeoSPARQL and Strabon are the most performant systems in this comparison. JenaGeoSPARQL scores most points for the two smaller datasets independent of caches. It also comes on top for the 1M triple dataset for spatial selection queries independent of caches. With cold caches and spatial selection queries it performs the best even in the largest dataset. Strabon on the other hand is the strongest competitor for warm cache executions and for spatial join queries on larger datasets regardless of caching. RDF4J also got a point for the spatial selection query with cold caches against the smallest 10K dataset but we note that it performs predictably through out the entire spectrum. Warm cache times are on average 10% 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 improvements over RDF4J only in the spatial join queries and for the largest dataset. Improvement ranges between 12% for the low selectivity spatial join SC2 and 42% for the higher selectivity join SC3. Thus the extra repository size and additional ingestion time for RDF4J+ maybe justified for specific use cases. Virtuoso performs adequately but its behaviour is not entirely predictable. While response times scale in an analog 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 the lowest response times for spatial joins with low selectivity such as SC2. Enabling the Lucene-based



(a) (b)

Figure 7 Response times – SC2 spatial join query.



(a) (b)

Figure 8 Response times – SC3 spatial join query.

GeoSPARQL plugin and using the standard spatial function querysets, the GraphDB+ offered borderline benefits to the worst scenarios with larger datasets. Using the spatial predicate version of the querysets, as instructed in the official documentation [58], 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 **x7**. However this negatively affected the times for SC1 and SC3 queries, in some cases by a factor of **(x7)**. This variant also provided small improvements in its sensitivity to caching. Using the *variance* column of `vquery_ordered_aggrs_4` database view on query response times, we can provide a few final notes concerning the consistency of the various systems tested. GraphDB+P on SC2 with cold caches and JenaGeoSPARQL on SC2 and SC3 irrespective of caches exhibited the highest variance especially in the two bigger 1M and 10M datasets. All other system, query, cache and dataset combinations had a consistent query response time across iterations.

The results of the above benchmark are available at Github [36] and Zenodo [35] for verification.

7.5.4 Experiment Summary

Considering all previous measurements (ingestion time, repository size and response time) we conclude that Virtuoso and JenaGeoSPARQL behave in a more balanced and predictable manner than the rest of the system configurations. Therefore they are well suited as a first choice for use cases with a mix of spatial workloads. JenaGeoSPARQL performs better in the three smaller datasets while Virtuoso scales better over 10M triple datasets.

For use cases where query response time is of essence, spatial datasets are well above 1M triples and relatively static, Strabon is a good candidate. This is because despite its slow ingestion times and large repository sizes its response times with warm caches are at least **100x** faster

than the second contender. This is in sync with the Geographica 2 GeoSPARQL benchmark's findings which affirmed Strabon's query response scalability with datasets up to 500M triples (approximately 50 million complex geometries).

For use cases where ingestion time and repository size are critical factors GraphDB with the GeoSPARQL plugin disabled is a perfect candidate. Its lower performance on low selectivity spatial joins for larger datasets can be mitigated because of the versatility of the GeoSPARQL plugin. If the need arises then the plugin can be enabled with simple DML SPARQL statements providing a choice between two spatial indexing methods (quad, geohash) and adjustable precision. The user can then use spatial predicates instead of spatial functions for the low selectivity spatial joins while using spatial functions in the rest of their queries. At no extra cost the user can disable the plugin without loosing the created spatial index.

RDF4J's Nativestore has improved its scalability over its predecessor OpenRDF Sesame as its excellent performance for the first three datasets does not degrade by a significant factor for the 10M triple dataset. Its results position it as a very good choice for use cases with low to medium size spatial dataset needs and its SAIL layered architecture as an ideal base RDF store for research projects which can innovate with more scalable techniques. Unfortunately, the optional Lucene SAIL's improvements in the largest dataset do not change RDF4J's position compared to the competitors. With this in mind, we believe it does not justify the added repository size and ingestion time it requires.

7.6 Benchmarking with Docker – Examples

The GeoRDFBench web site [32] contains detailed tutorials and 4 Docker examples [3.4.4]. 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 Appendix D Table 13.

7.7 Benchmarking Step-By-Step

In this part we will go through the docker[21] based example *"RDF4J, GraphDB and JenaGeoSPARQL against Scalability-10K, 100K, 1M Workloads"*. Basically, this is a subset of the experiments presented earlier. We will outline the steps taken and elaborate on some important aspects behind the scenes that might be of interest to users that would like to create their own custom benchmarks.

1. *Pulling the image*: The GeoRDFBench's github repository has a docker registry from which we can pull the image *tioannid/geordfbench/multistore/scal10k_1m*.

```
$ docker pull ghcr.io/tioannid/geordfbench/multistore/scal10k_1m:latest
```

[21] A computer with Docker installation is required.

2. *Starting the container*: From the pulled image a container named *multiscal10k1m* can be started. We provide the hostname, IP address, RAM and number of virtual CPUs assigned to the container. We provide a mapping of the current host's working folder to the container's `/src/` folder. Finally we expose the container's PostgreSQL listening port 5432 to the physical host's 5432 port and provide PostgreSQL's standard authentication credentials.

```
$ docker run -d -e POSTGRES_PASSWORD=postgres -p 5432:5432/tcp --hostname NUC8i7BEH \
--cpus="4" --memory="11g" --memory-swap="11g" \
--mount "type=bind,src=‘pwd‘,target=/src" --name multiscal10k1m \
 ghcr.io/tioannid/geordfbench/multistore/scal10k_1m:latest
```

3. *Launch the experiments*: Through a terminal to the running container the user can launch the experiments and watch the creation of repos and experiment execution for each RDF store and each one of the 3 workloads.

```
$ docker exec -it multiscal10k1m /bin/bash
root@NUC8i7BEH:/data# ./startUpScript.sh
```

7.7.1 Docker Image Contents

7.7.1.1 Dockerfile

The docker image is built based on a *Dockerfile* which describes and installs the software prerequisites and takes any actions required to setup database server configuration and create any custom filesystem layout. The particular image installs Ubuntu Jammy, JDK 11, PostgreSQL 14 with its dependencies, Maven and Git. The actions it takes are:

a. Setups a *systemd service* for PostgreSQL
b. Copies under the container's `/` folder the `startUpScript.sh`
c. Copies under the container's `/data` folder the following data:
   i. GeoRDFBench Framework's source and binary code, under `/data/geordfbench`
   ii. The distribution binaries of the RDF stores to be tested, under `/data/compressed`
   iii. The RDF datasets required, under `/data/Geographica2_Datasets`
   iv. Scripts to create the database for custom reports and statistics and run the experiments (repo creation, workload execution) for each RDF store, under the `/data/scripts`

7.7.1.2 Start Up Script

This is a high level wrapper script that instruments the steps of the experiments' execution. It is also the best place to start understanding the experiment process in a top-down fashion. The script takes the following actions:

a. Starts PostgreSQL service and create the database schema for custom reports and experiment statistics to be stored.
b. For each RDF store calls a specific experiment script in `/data/scripts/run*.sh`

7.7.1.3 Experiment Execution Script

This is an script specific for each system, however it has the same logic which is summarized in the following steps:

a. The environment preparation script `geordfbench/scripts/prepareRunEnvironment.sh` is called, which also prints and logs the environment variables set for the system.
b. With the repository generation wrapper script creates the three Scalability repositories
c. Invokes three times the compact workload run script of the system against each one of the Scalability workloads.

7.7.1.4 Inspection Of Logs

The experiment execution script placed the logs under each system's base directory, but could very easily directed them at a custom location for centralized collection. In this case however the repository creation logs can be found in `/data/geordfbench/*SUT/scripts/CreateRepos` while the workload execution logs in `/data/geordfbench/*SUT/scripts/RunTests3`.

7.7.1.5 Inspection Of Standard Statistics

The standard file-system based statistics that GeoRDFBench generates by default were placed in `/data/Results_Store` and the following sample listing clarifies the layout:

```
root@NUC8i7BEH:/data# tree -L 2 Results_Store/
Results_Store/
|-- RDF4JSUT
    |-- 2025-05-16_RDF4JSUT_RunWL_Scal100K
    |-- 2025-05-16_RDF4JSUT_RunWL_Scal10K
    `-- 2025-05-16_RDF4JSUT_RunWL_Scal1M
        `-- Scalability
            `-- 1M
                `-- RDF4JSUT-ExperimentWorkload
                    |-- 00-SC1_Geometries_Intersects_GivenPolygon-cold
                    |-- 00-SC1_Geometries_Intersects_GivenPolygon-cold-long
                    |-- 00-SC1_Geometries_Intersects_GivenPolygon-warm
                    |-- 00-SC1_Geometries_Intersects_GivenPolygon-warm-long
...
```

7.7.1.6 Inspection Of Custom Report Store

The PostgreSQL database has been used as the report sink and all detailed experiment statistics have been sent to it. From the launched terminal that the startup script was run, we connect to the database and can query the basic reporting views. The `vqueryexecution3` shows detailed information (21 columns) about each iteration of query execution (iteration): the experiment, host, sut, queryset, dataset, query label, query no, cache type, query iteration, evaluation time, scan time, no of results, no of scan errors, evaluation flag, result exception, total time = evaluation + scan time, total time in seconds, was execution valid flag. The `vquery_ordered_aggrs_3` aggregates the above information (11 columns) and for query execution shows: experiment id, sut, queryset, dataset, query label, query no, was execution valid flag, cache type, no of iterations, mean of total time, median of total time. Users are free to modify the aggregate views in order to customize them to their needs e.g., adding more aggregate functions like variance and standard deviation for the total time or some other quantity collected during the experiments e.g., no of results.

```
root@NUC8i7BEH:/data# su - postgres
postgres@NUC8i7BEH:~$ psql
psql (14.17 (Ubuntu 14.17-0ubuntu0.22.04.1))
Type "help" for help.

postgres=# \c geordfbench
You are now connected to database "geordfbench" as user "postgres".
geordfbench=# SELECT * FROM public.vqueryexecution3;
geordfbench=# SELECT * FROM public.vquery_ordered_aggrs_3;
```

Table 7 Benchmarking Coverage [3.1].

Criteria	1. RDF Data Ingestion (Loading and Indexing)	2. Query Execution Performance and Accuracy	3. RDF Store Data Scalability and User Concurrency	4. Statistics Reporting and Logging
GeoRDFBench Benchmarking Framework	Yes, choice of best loader, split time and size for repo loading and indexing	Yes, split query execution time for evaluation and scan phases, query fine error handling, query accuracy verification, cold/warm cache execution, continuous queryset execution	Yes, scalability workload (500M-triples 50M geometries), scalable synthetic data and query generator, simulated concurrent user execution	Yes, repo generation and query execution logs, system and host environment logs, sampling resultsets, customized report sinks (database and filesystem), custom statistics measurements, ground query generation
LITMUS Benchmarking Framework	Partial, repository final size not reported	Partial, no result accuracy, no query filtering, no query namespace prefixes handling, no query execution error handling/reporting, cold/warm execution	No, small datasets readily available	Partial, file based reports and logs, no alternate report sink, plotting script available
IGUANA Benchmarking Framework	Partial, repository final size and load time not reported, HTTP endpoints only	Adequate, evaluation from scan time cannot be distinguished, no result accuracy, query evaluation and scan errors not handled appropriately, continuous/warm execution	Adequate, medium size datasets DBpedia 3.5.1 (232.5M triples) and SWDF (294.8K triples)	High, results stored as RDF or CSV files or endpoints, multiple statistics measurements
HOBBIT Benchmarking Platform	No, user implemented	No, user implemented, warm cache execution, no query execution error handling/reporting	No, user provided	Yes
Geographica 2 GeoSPARQL Benchmark	Adequate	Adequate, cold/warm cache execution, continuous queryset execution	High, scalability workload (500M-triples 50M geometries), scalable synthetic data and query generator	Adequate, repo generation and query execution logs, system and host environment logs, sampling resultsets, filesystem report and logs

8 Comparisons of GeoRDFBench

In this section we present key features, limitations and current level of support for the benchmarking frameworks LITMUS, IGUANA, the Geographica 2 benchmark and the benchmarking platform HOBBIT. This will help the reader understand some of the key contributions of GeoRDFBench over each one of the above solutions. In order to streamline this process, when relevant, the comparison criteria from Section 3 are referenced by number in square brackets after the relevant text. Tables 7-11 map each benchmarking solution's coverage (strength) of Section's 3 criteria.

8.1 LITMUS Benchmarking Framework

LITMUS is an early effort of creating a framework which aims at uniform benchmarking of data management solutions (DMS) supporting different query languages, such as SPARQL for RDF stores, Gremlin for LPG stores and SQL for traditional DBMSs. Its deliverables are: (i) *Gremlinator* [76], a SPARQL-to-Gremlin query translator, (ii) the `litmus-docker` [76] image for running experiments on 4 LPGs systems, 4 RDF stores and plotting the results and (iii) a Python utility which converts the SPARQL query JSON resultset from a DBpedia endpoint to GraphML.

It can be classified as a bare bones framework since it provides a basic level of automation with a collection of scripts, organized by system, for loading data and submitting queries in various languages such as Bash, Python and Gremlin Groovy. The main entry point is a large (approximately 2000 lines) monolithic Python script *run_script.py* where magic literals are used extensively e.g., LPG system folders are prepended with "g_", RFD stores with "r_" and system scripts have to follow a rigid naming convention to distinguish actions that relate to them. There is no common engine to implement execution logic variants e.g., easily changing the order and number of cold or warm cache executions, handle queryset, dataset and/or system specific namespace prefixes. Instead there is one script for each system and benchmarking operation to test, however

Table 8 Geospatial Support [3.2].

Criteria	1. Spatial Index Creation Time and Size	2. Spatial Index Specification	3. GeoSPARQL Support and Compliance	4. GeoSPARQL Spatial Indexing Challenges	5. Spatial Index Enabled GeoSPARQL Query Execution
GeoRDFBench Benchmarking Framework	Yes	Yes, choice of method, accuracy and time of index creation	Yes + query translation for system-specific vocabularies	Yes	Yes
LITMUS Benchmarking Framework	No	No	No	No	No
IGUANA Benchmarking Framework	No	No	No	No	No
HOBBIT Benchmarking Platform	No, user provided	No, user provided	No, user provided	No, user provided	No, user provided
Geographica 2 GeoSPARQL Benchmark	Yes	Yes, choice of method, accuracy and time of index creation	Yes + query translation for system-specific vocabularies	Yes	Partial

many of them are trivial (2-4 lines), repetitive (see each system's cold and warm cache scripts) and offer a primitive parameterized wrapper script over each systems' own utilities e.g., bulk loaders. Finally, there is a script for plotting the loading and execution times.

The "Parameters Measured" portion of the github `litmus-docker` repository mentions a large number of low-level (CPU/PMU) hardware level metrics for which there is no evidence of what practical use they offer in the graph store benchmarking evaluation. This is done by virtue of using the Linux profiler `perf` Python package in specific cloned scripts. Since this package explicitly states that it runs only on Linux[22] this basically implies that running LITMUS natively, for example in Windows and not in a docker container, will not have its full functionality.

The criterion category [3.1] "Benchmarking Coverage" is partially covered because of these missing features: (i) *repository final size is not reported*, (ii) there is no way to differentiate query evaluation time from scan time, (iii) there is *no result accuracy checking* measurement, (iv) there is *no query inclusion/exclusion filters* for running query subsets, (v) *cannot handle namespace prefixes at the system, queryset or dataset level*, (vi) query evaluation and/or scan errors are not reported, (vii) there is no active error handling and no related policy to respond to them, (viii) scalability cannot be tested with the available small datasets, (ix) all results are file based and no alternate report source is offered. The [3.2] "Geospatial Support" is not covered at all. Quads are not supported and since N-Triples and Turtle are the only supported RDF serializations only the criterion [3.3.2] from "Semantic Web Standards Support" is partially covered. The criterion category [3.4] "Framework Automation Level" is covered at a basic level since: (i) *query templates are not supported*, (ii) there is no common engine but a large monolithic entry point script, with multiple system relevant trivial and repetitive scripts using magic literals and (iii) it only supports *privileged dockerized operation* with Linux OS due to the use of the `perf` Python package. As a result setting up or updating benchmarks and adapting experiments requires extensive and meticulous modifications to all relevant system scripts in order to preserve integrity.

Finally, the [3.5] "Sustainability and Extensibility" criterion category is very weakly supported. The resource *is not sustainable* because: (i) the framework is not actively maintained and updated, as the paper was published in 2017 and there no commits for the past 8 years, (ii) the systems tested are deprecated e.g., Apache Jena v3.2.0 (released, Feb 2017), OrientDB 2.1.3 (released, October 2015), etc., (iii) the docker image fails to be buit on a standard Linux Ubuntu 22.04.5 LTS, and (iv) the advent of the standard GQL for LPGs basically suggests that the supporting group's research efforts should probably need to target a SPARQL-to-GQL converter.

8.2 IGUANA Benchmarking Framework

IGUANA aims to be a generic framework for benchmarking the read-write performance of triple stores. In its current status [26], it supports benchmarking SPARQL HTTP endpoints. It addition to querying it also allows for updates and stress testing endpoints by simulating users sending sets

[22] `https://docs.python.org/3/howto/perf_profiling.html`

Table 9 Semantic Web Standards Support [3.3].

Criteria	1. Quads as well as Triples	2. Multiple RDF Serialization Support	3. SPARQL Query Namespace Prefixes
GeoRDFBench Benchmarking Framework	Yes	Yes	Yes, dynamic namespace prefix map merging
LITMUS Benchmarking Framework	No	Partial, N-Triples, Turtle	No
IGUANA Benchmarking Framework	Yes	Yes	No
HOBBIT Benchmarking Platform	No, user provided	No, user provided	No, user provided
Geographica 2 GeoSPARQL Benchmark	Yes	Yes	Adequate, non-dynamic namespace prefix map merging

of queries independently of each other. It features a YAML file where all benchmark configuration is stored. This file contains the connections to RDF store endpoints, the description of tasks to run against endpoints, the metrics to measure and the destinations (CSV, RDF, endpoint) for report statistics. Its written in Java and is offered in two forms: (i) a Java jar file, and (ii) a native executable produced by compiling with GraalVM[23]. Despite its solid inception and implementation, focusing on SPARQL endpoint benchmarking bears a toll on the features IGUANA can support.

The criterion category [3.1] "Benchmarking Coverage" is adequately covered with some missing features: (i) only operates on HTTP endpoints, (ii) *repository final size is not reported* (iii) it *does not measure repository load times* (bulk loaders do not operate over HTTP), (iv) there is no way to differentiate query evaluation time from scan time, (v) there is *no result accuracy checking* measurement, (vi) query evaluation and/or scan errors are aggregated under "unknownException" statistics column along with HTTP related error codes, (vii) there can be no active error handling and no related policy to respond to them (e.g., ignore them and continue), (viii) scalability can borderline be tested with the available average size (232.5M triples) datasets. The [3.2] "Geospatial Support" is not covered at all. The criterion category [3.3] "Semantic Web Standards Support" is covered with the exception of criterion [3.3.3]. The criterion category [3.4] "Framework Automation Level" is covered at an adequate-high level because of its solid inception and implementation and the use of YAML specs. Missing features include: (i) query templates support is not generic but accommodates a specific scenario where query parameters are replaced by values returned from querying other live endpoints with related data, (ii) there is *no query inclusion/exclusion filters* for running query subsets, and (iii) provides native but not dockerized execution.

Finally, the [3.5] "Sustainability and Extensibility" criterion category is of medium strength. The resource seems to be borderline sustainable as it has active commits during the last 18 months and uses high quality libraries and techniques. It also seems to be quite easy to extend it by introducing new endpoints and workloads for testing, however there is no easy way to extend it in direct testing of GeoSPARQL/SPARQL engines through an RDF client library.

8.3 HOBBIT Benchmarking Platform

HOBBIT [70] 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.

It has been used to benchmark non-LD applications and, among other things, allows it to run benchmarks throughout the LD life-cycle [73, 41]. This flexibility is achieved by containerizing the benchmarking components and using the RabbitMQ [48] message broker, which allows *byte array* "messages" to be exchanged between them. This *stripped from semantics representation* is exactly

[23] `https://www.graalvm.org/`

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 [30]. 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* [72] provides some evidence [24] 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.

In our view, this framework *targets a completely different user group* than the other frameworks and benchmarks in this work. 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[25].

The user of other benchmarking solutions to a lesser or greater (GeoRDFBench) extent is mainly interested in benchmarking RDF store performance in terms of data loading, data indexing, query accuracy and query execution times. They have expertise in Java or Python 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 some level of 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 this framework is not directly competitive* with the other solutions and a direct comparison with quantitative criteria is neither possible nor useful.

From the criterion categories [3.1] "Benchmarking Coverage", [3.2] "Geospatial Support" and [3.3] "Semantic Web Standards Support" the only criterion HOBBIT provides explicit support for is [3.1.4] "Statistics Reporting and Logging". For every other criterion the user has to provide the implementation. "Framework Automation Level" category [3.4] is almost fully covered with a note that only dockerized execution is available which implies that cold cache executions are difficult to have. Finally, for the [3.5] "Sustainability and Extensibility" criterion category we verify that it is highly sustainable since it is actively maintained and supported with quality libraries, however, it is not easy to extend its ontology and APIs for directly supporting SPARQL/GeoSPARQL data loading and querying benchmarking.

[24] `https://github.com/hobbit-project/java-sdk-example/blob/master/SDK_vs_Standard_Way.pdf`
[25] `https://hobbit-project.github.io/challenge_overview.html`

Table 10 Framework Automation Level [3.4].

Criteria	1. Setting Up New Benchmarks	2. Updating Existing Benchmarks	3. Repeatable and Adaptable Experiment Execution	4. Native and Dockerized Execution
GeoRDFBench Benchmarking Framework	Yes, scalable GeoSPARQL data and query generator, template queries	Yes, declarative specs, reusability through class inheritance and script hierarchies, maven multi-module POM project	Yes, benchmarking environment specs, query filtering at execution	Yes, both on Linux and Windows OSs
LITMUS Benchmarking Framework	Basic, large monolithic entry point script, trivial repetitive scripts, magic literals, no common engine, ground queries only	Weak, updates require extensive and risky script modifications	Weak, adapting requires extensive and risky script modifications	Adequate, privileged dockerized operation only, Linux only due to 'perf' package
IGUANA Benchmarking Framework	Yes, non generic query templates	High, YAML specs, solid inception and implementation	Adequate, YAML benchmark specs, no query filtering	Adequate, native execution but with easy dockerization
HOBBIT Benchmarking Platform	Yes	High, ontology based benchmark and system specs	High	Adequate, dockerized execution only
Geographica 2 GeoSPARQL Benchmark	High, scalable GeoSPARQL data and query generator	Partial, reusability through class inheritance and basic scripts	Weak, medium effort for updating benchmarks and systems	Adequate, native execution only

Table 11 Sustainability and Extensibility [3.5].

Criteria	Sustainability and Extensibility
GeoRDFBench Benchmarking Framework	Yes, actively maintained and supported, quality libraries, code base designed and implemented on OO principles
LITMUS Benchmarking Framework	Weak, last update 8 years ago, extending requires multiple risky script modifications
IGUANA Benchmarking Framework	Medium, active commits in the last 18 months, high quality libraries, easy to extend with new endpoints, difficult to extend for non HTTP benchmarking
HOBBIT Benchmarking Platform	High, actively maintained and supported, quality libraries, difficult to extend ontology and APIs for SPARQL/GeoSPARQL cases
Geographica 2 GeoSPARQL Benchmark	Medium, supported but not actively maintained, medium effort for updating benchmarks and systems

8.4 Geographica 2 GeoSPARQL Benchmark

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 [14, 11] 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 [14, 11] was limited to workload-specific queryset and experiment class hierarchies.

Table 12 SLOC Comparison with Geographica 2.

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

GeoRDFBench expands the scope of Universe of Discourse (UoD) [14] 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)* [56, 47]. More specifically, the utility **cloc** [18] measured the number of files and the *physical* SLOCs for the runtime and RDF modules for the

two systems. Table 12 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.

With respect to compliance to the criteria categories in Section 3 we an adequate-high rating. The [3.1] "Benchmarking Coverage" is adequately covered with some missing features: (i) there is *no result accuracy checking* measurement, (ii) no query fine error handling. Scalability though is thoroughly provisioned with scalability workload and a scalable synthetic data and query generator. The [3.2] "Geospatial Support" is almost entirely covered with minor issues such as the limitation of query translation to specific systems and vocabularies and the lack of spatial predicate versions of querysets. The criterion category [3.3] "Semantic Web Standards Support" is covered almost entirely with the exception of non-dynamic namespace prefix map merging. The criterion category [3.4] "Framework Automation Level" is covered at an adequate level. Missing features include: (i) query templates support is not generic and not fully automated, (ii) limited reusability through class inheritance and basic script structure, (iii) provides native execution only, (iv) there is *no query inclusion/exclusion filters* for running query subsets, and (v) updating benchmark workloads and systems requires medium effort to complete. Finally, for the [3.5] "Sustainability and Extensibility" criterion category we verify that it is not very sustainable since it is supported but not actively maintained. Also, extending the solution requires medium to high effort to achieve.

9 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 [31, 33] and site [32].

Future work will include: (i) implementing the remaining *Create, Update and Delete operations of the JSON Library endpoint* module, (ii) a *user interface module* paired with the endpoint which will assist with generating and modifying JSON specifications, (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** [9] can be tested.

References

1 Marcel R Ackermann, Hannah Bast, Benedikt Maria Beckermann, Johannes Kalmbach, Patrick Neises, and Stefan Ollinger. The dblp knowledge graph and sparql endpoint. *Transactions on Graph Data and Knowledge*, 2(2):3:1–3:23, 2024. doi:10.4230/TGDK.2.2.3.

2 Altair. AnzoGraph DB web site, 2025. URL: https://docs.cambridgesemantics.com/anzograph/v2.5/userdoc/Home.htm.

3 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. G-CORE: A core for future graph query languages. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1421–1432. ACM, 2018. doi:10.1145/3183713.3190654.

4 Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. Rdf and property graphs interoperability: Status and issues. *AMW*, 2369, 2019. URL: https://ceur-ws.org/Vol-2369/paper01.pdf.

5 Hannah Bast, Patrick Brosi, and Johannes Kalmbach. Efficient spatial joins on large geometry sets. In *Proceedings of the 33rd ACM International Conference on Advances in Geographic Information Systems*, pages 466–476, 2025. doi:10.1145/3748636.3762757.

6 Hannah Bast, Johannes Kalmbach, Christoph Ullinger, and Robin Textor-Falconi. Sparqloscope: A generic benchmark for comprehensive performance evaluation of sparql engines, May 2025. doi:10.5281/zenodo.15387492.

7 Robert Battle and Dave Kolas. Enabling the geospatial Semantic Web with Parliament and GeoSPARQL. *Semantic Web*, 3(4):355–370, 2012. doi:10.3233/SW-2012-0065.

8 Pierfrancesco Bellini and Paolo Nesi. Performance assessment of RDF graph databases for smart city services. *J. Vis. Lang. Comput.*, 45:24–38, 2018. doi:10.1016/j.jvlc.2018.03.002.

9 Dimitris Bilidas, Theofilos Ioannidis, Nikos Mamoulis, and Manolis Koubarakis. Strabo 2: Distributed management of massive geospatial rdf datasets. In *The Semantic Web–ISWC 2022: 21st International Semantic Web Conference, Virtual Event, October 23–27, 2022, Proceedings*, pages 411–427. Springer, 2022. doi:10.1007/978-3-031-19433-7_24.

10 Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009. doi:10.4018/JSWIS.2009040101.

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

12 European Commission's Joint Research Centre. INSPIRE Directive web site, 2007. URL: https://inspire.ec.europa.eu/inspire-directive/2.

13 Hirokazu Chiba, Ryota Yamanaka, and Shota Matsumoto. G2gml: Graph to graph mapping language for bridging rdf and property graphs. In *The Semantic Web–ISWC 2020: 19th International Semantic Web Conference, Athens, Greece, November 2–6, 2020, Proceedings, Part II 19*, pages 160–175. Springer, 2020. doi:10.1007/978-3-030-62466-8_11.

14 Peter Coad, Edward Yourdon, et al. *Object-oriented analysis*, volume 2. Yourdon press New York, 1992.

15 European Commission. SWING Project web site, 2006. URL: https://cordis.europa.eu/project/id/026514.

16 European Commission. ExtremeEarth Project web site, 2019. URL: http://earthanalytics.eu/.

17 Lixi Conrads, Jens Lehmann, Muhammad Saleem, Mohamed Morsey, and Axel-Cyrille Ngonga Ngomo. Iguana: A generic framework for benchmarking the read-write performance of triple stores. In Claudia d'Amato, Miriam Fernandez, Valentina Tamma, Freddy Lecue, Philippe Cudré-Mauroux, Juan Sequeda, Christoph Lange, and Jeff Heflin, editors, *The Semantic Web – ISWC 2017*, pages 48–65, Cham, 2017. Springer International Publishing.

18 Albert Danial. cloc: v1.92 source code on Github, 2021. URL: https://github.com/AlDanial/cloc.

19 Alishiba Dsouza, Nicolas Tempelmeier, Ran Yu, Simon Gottschalk, and Elena Demidova. WorldKG: A world-scale geographic knowledge graph. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, pages 4475–4484. ACM, 2021. doi:10.1145/3459637.3482023.

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

21 Orri Erling and Ivan Mikhailov. Virtuoso: Rdf support in a native rdbms. In *Semantic web information management: a model-based perspective*, pages 501–519. Springer, 2009. doi:10.1007/978-3-642-04329-1_21.

22 FasterXML. FasterXML Jackson source code on Github, 2024. URL: https://github.com/FasterXML/jackson.

23 Raphael A Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4:1–9, 1974. doi:10.1007/BF00288933.

24 Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pages 1433–1445, 2018. doi:10.1145/3183713.3190657.

25 George Garbis, Kostis Kyzirakos, and Manolis Koubarakis. Geographica: A benchmark for geospatial rdf stores (long version). In *The Semantic Web–ISWC 2013: 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II 12*, pages 343–359. Springer, 2013. doi:10.1007/978-3-642-41338-4_22.

26 DICE Group. Iguana source code on Github, 2024. URL: https://github.com/dice-group/IGUANA.

27 Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005. doi:10.1016/J.WEBSEM.2005.06.005.

28 Olaf Hartig. Foundations of rdf* and sparql*:(an alternative approach to statement-level metadata in rdf). In *AMW 2017 11th Alberto Mendelzon*

International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017., volume 1912. Juan Reutter, Divesh Srivastava, 2017.

29 Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 562–573. Morgan Kaufmann, 1995. URL: http://www.vldb.org/conf/1995/P562.PDF.

30 HOBBIT. HOBBIT General API, 2017. URL: https://hobbit-project.github.io/platform_api.html.

31 Theofilos Ioannidis. GeoRDFBench Framework. Software, version 2.0.0, swhId: swh:1:dir:972135253c56ac42c3face1b01ab2ad360acea96 (visited on 2026-08-12). URL: https://github.com/tioannid/geordfbench, doi:10.4230/artifacts.27628.

32 Theofilos Ioannidis. GeoRDFBench Framework Web Site. Service, version 2.0.0. (visited on 2026-08-12). URL: https://geordfbench.di.uoa.gr/, doi:10.4230/artifacts.27629.

33 Theofilos Ioannidis. GeoRDFBench Framework Samples source code on Github, 2023. URL: https://github.com/tioannid/geordfbench_samples.

34 Theofilos Ioannidis. Geographica 2 Dataset doi, August 2024. doi:10.5281/zenodo.13283414.

35 Theofilos Ioannidis. GeoRDFBench Framework Benchmark Results doi, May 2025. doi:10.5281/zenodo.15349539.

36 Theofilos Ioannidis. GeoRDFBench Framework Benchmark Results on Github, 2025. URL: https://github.com/tioannid/geordfbench_results.

37 Theofilos Ioannidis, George Garbis, Kostis Kyzirakos, Konstantina Bereta, and Manolis Koubarakis. Evaluating geospatial rdf stores using the benchmark geographica 2. *Journal on Data Semantics*, 10(3-4):189–228, 2021. doi:10.1007/S13740-021-00118-X.

38 ISO/IEC. Information technology - Database languages - GQL, April 2024. URL: https://www.iso.org/standard/76120.html.

39 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. Know, know where, knowwheregraph: A densely connected, cross-domain knowledge graph and geo-enrichment service stack for applications in environmental intelligence. *AI Mag.*, 43(1):30–39, 2022. doi:10.1609/aimag.v43i1.19120.

40 Ernesto Jiménez-Ruiz, Tzanina Saveta, Ondrej Zamazal, Sven Hertling, Michael Roder, Irini Fundulaki, Axel Ngonga Ngomo, Mohamed Sherif, Amina Annane, Zohra Bellahsene, et al. Introducing the hobbit platform into the ontology alignment evaluation campaign. In *13th International Workshop on Ontology Matching (OM)*, volume 2288, pages 49–60, 2018. URL: https://ceur-ws.org/Vol-2288/om2018_LTpaper5.pdf.

41 Milos Jovanovik. HOBBIT Examples source code on Github, 2017. URL: https://github.com/hobbit-project/SpatialBenchmark.

42 Milos Jovanovik, Timo Homburg, and Mirko Spasić. A geosparql compliance benchmark. *ISPRS International Journal of Geo-Information*, 10(7):487, 2021. doi:10.3390/IJGI10070487.

43 Nikolaos Karalis, Georgios M. Mandilaras, and Manolis Koubarakis. Extending the YAGO2 knowledge graph with precise geospatial knowledge. In Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon, editors, *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part II*, volume 11779 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2019. doi:10.1007/978-3-030-30796-7_12.

44 Charalampos Kostopoulos, Giannis Mouchakis, Antonis Troumpoukis, Nefeli Prokopaki-Kostopoulou, Angelos Charalambidis, and Stasinos Konstantopoulos. Kobe: Cloud-native open benchmarking engine for federated query processors. In *The Semantic Web: 18th International Conference, ESWC 2021, Virtual Event, June 6–10, 2021, Proceedings 18*, pages 664–679. Springer, 2021. doi:10.1007/978-3-030-77385-4_40.

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

46 Kostis Kyzirakos, Manos Karpathiotakis, and Manolis Koubarakis. Strabon: A semantic geospatial dbms. In *The Semantic Web–ISWC 2012: 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I 11*, pages 295–311. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-35176-1_19.

47 Davy Landman, Alexander Serebrenik, and Jurgen Vinju. 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*, pages 221–230, 2014. doi:10.1109/ICSME.2014.44.

48 Rabbit Technologies Ltd. RabbitMQ Broker web site, 2007. URL: https://www.rabbitmq.com/.

49 Manolis Koubarakis, editor. *Geospatial data science: a hands-on approach based on geospatial technologies*. ACM Books, 2023.

50 Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. Dbpedia sparql benchmark–performance assessment with real queries on real data. In *The Semantic Web–ISWC 2011: 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I 10*, pages 454–469. Springer, 2011. doi:10.1007/978-3-642-25073-6_29.

51 Thomas Mueller. H2 Database) web site, 2005. URL: https://h2database.com/.

52 MvnRepository. Maven Central Repository web site, 2024. URL: https://mvnrepository.com/repos/central.

53 neo4j. Neo4j Spatial source code on Github, 2025. URL: https://github.com/neo4j-contrib/spatial.

54 neosemantics. neosemantics (n10s) source code on Github, 2025. URL: https://github.com/neo4j-labs/neosemantics.

55 Axel-Cyrille Ngonga Ngomo, Sören Auer, Jens Lehmann, and Amrapali Zaveri. Introduction to linked data and its lifecycle on the web. *Reasoning Web. Reasoning on the Web in the Big Data Era: 10th International Summer School 2014, Athens, Greece, September 8-13, 2014. Proceedings 10*, pages 1–99, 2014. doi:10.1007/978-3-319-10587-1_1.

56 Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A sloc counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16. Citeseer, 2007.

57 Ontotext. GraphDB, 2024. URL: https://graphdb.ontotext.com/.

58 Ontotext. GraphDB geosparql, 2024. URL: https://graphdb.ontotext.com/documentation/10.7/geosparql-support.html#geosparql-support.

59 Ontotext. GraphDB ImportRDF geosparql, 2024. URL: https://graphdb.ontotext.com/documentation/10.8/command-line-tools.html#importrdf.

60 Oracle. Spatial And Graph web site, 2019. URL: https://docs.oracle.com/en/database/oracle/oracle-database/19/spatial-and-graph.html.

61 Taha Osman and Gregory Albiston. Geosparql-jena: Implementation and benchmarking of a geosparql graphstore. In *23rd European Conference on Knowledge Management Vol 2*. Academic Conferences and publishing limited, 2022.

62 Alisdair Owens, Andy Seaborne, Nick Gibbins, and m.c Schraefel. Clustered tdb: A clustered triple store for jena, November 2008.

63 Kostas Patroumpas, Giorgos Giannopoulos, and Spiros Athanasiou. Towards geospatial semantic data management: strengths, weaknesses, and challenges ahead. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 301–310, 2014. doi:10.1145/2666310.2666410.

64 Matthew Perry, Ana Estrada, Souripriya Das, and Jayanta Banerjee. Developing geosparql applications with oracle spatial and graph. In *SSN-TC/OrdRing@ ISWC*, pages 57–61, 2015. URL: https://ceur-ws.org/Vol-1488/paper-06.pdf.

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

66 Suprio Ray, Bogdan Simion, and Angela Demke Brown. Jackpine: A benchmark to evaluate spatial database performance. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1139–1150. IEEE, 2011. doi:10.1109/ICDE.2011.5767929.

67 RDF4J. RDF4J, 2024. URL: https://rdf4j.org/.

68 RDF4J. RDF4J lucene geosparql, 2024. URL: https://rdf4j.org/documentation/programming/geosparql/.

69 RDF4J. RDF4J NativeStore API Doc, 2024. URL: https://rdf4j.org/javadoc/4.3.16/org/eclipse/rdf4j/sail/nativerdf/NativeStore.html.

70 Michael Röder, Denis Kuchelev, and Axel-Cyrille Ngonga Ngomo. HOBBIT: A platform for benchmarking big linked data. *Data Sci.*, 3(1):15–35, 2020. doi:10.3233/ds-190021.

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

72 Michael Röder. HOBBIT Java SDK web site, 2018. URL: https://github.com/hobbit-project/java-sdk.

73 Michael Röder. HOBBIT Examples source code on Github, 2024. URL: https://github.com/hobbit-project/hobbit.examples.

74 Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. Feasible: A feature-based sparql benchmark generation framework. In *The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I 14*, pages 52–69. Springer, 2015. doi:10.1007/978-3-319-25007-6_4.

75 Stardog. Stardog, 2024. URL: https://www.stardog.com/.

76 Harsh Thakkar. litmus-docker source code on Github, 2017. URL: https://github.com/LITMUS-Benchmark-Suite/litmus-docker.

77 Harsh Thakkar. Towards an open extensible framework for empirical benchmarking of data management solutions: Litmus. In *The Semantic Web: 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28–June 1, 2017, Proceedings, Part II 14*, pages 256–266. Springer, 2017. doi:10.1007/978-3-319-58451-5_20.

78 Antonis Troumpoukis, Stasinos Konstantopoulos, Giannis Mouchakis, Nefeli Prokopaki-Kostopoulou, Claudia Paris, Lorenzo Bruzzone, Despina-Athanasia Pantazi, and Manolis Koubarakis. Geofedbench: A benchmark for federated geosparql query processors. In *ISWC (Demos/Industry)*, pages 228–232, 2020. URL: https://ceur-ws.org/Vol-2721/paper558.pdf.

79 Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: a property graph query language. In Peter A. Boncz and Josep Lluís Larriba-Pey, editors, *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, page 7. ACM, 2016. doi:10.1145/2960414.2960421.

80 W3C. SPARQL 1.1 Query Language web site, 2013. URL: https://www.w3.org/TR/sparql11-query/.

81 Wikipedia. Geohash wiki page, 2008. URL: https://en.wikipedia.org/wiki/Geohash.

A Scalability Workload Serialized JSON Specifications

This appendix presents the detailed form of the three most important JSON specifications for the Scalability 10K workload as they were generated by the JSON generator API code samples in Section 6. Listings 11, 12, 13 show the serialized representations of the specifications for the Scalability 10K complex dataset, the spatial functions variant of the Scalability queryset and the Scalability experiment execution respectively.

Listing 11 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}
```

Listing 12 Serialized Queryset Specification.

```
{ "classname" : "gr.uoa.di.rdf.Geographica3.runtime.querysets.complex.impl.StaticTempParamQS",
"name" : "scalabilityFunc",
"relativeBaseDir" : "",
"hasPredicateQueriesAlso" : false,
"mapQueries" : {
"0" : {
"label" : "SC1_Geometries_Intersects_GivenPolygon",
"text" : "SELECT ?s1 ?o1 WHERE { \n ?s1 geo:asWKT ?o1 . \n FILTER(geof:FUNCTION(?o1, GIVEN_SPATIAL_LITERAL)).
    \n} \n",
"usePredicate" : false,
"expectedResults" : -1
},
"1" : {
"label" : "SC2_Intensive_Geometries_Intersect_Geometries",
"text" : "SELECT ?s1 ?s2 \nWHERE { \n ?s1 geo:hasGeometry [ geo:asWKT ?o1 ] ;\n lgo:has_code
    \"1001\"^^xsd:integer . \n ?s2 geo:hasGeometry [ geo:asWKT ?o2 ] ;\n lgo:has_code ?code2 . \n
    FILTER(?code2>5000 && ?code2<6000 && ?code2 != 5260) .\n FILTER(geof:FUNCTION(?o1, ?o2)). \n} \n",
"usePredicate" : false,
"expectedResults" : -1
},
"2" : {
"label" : "SC3_Relaxed_Geometries_Intersect_Geometries",
"text" : "SELECT ?s1 ?s2\nWHERE {\n ?s1 geo:hasGeometry [ geo:asWKT ?o1 ] ;\n lgo:has_code
    \"1001\"^^xsd:integer .\n ?s2 geo:hasGeometry [ geo:asWKT ?o2 ] ;\n lgo:has_code ?code2 .\n FILTER(?code2
    IN (5622, 5601, 5641, 5621, 5661)) .\n FILTER(geof:FUNCTION(?o1, ?o2)).\n} \n",
"usePredicate" : false,
"expectedResults" : -1
}
},
"mapUsefulNamespacePrefixes" : { },
```

```
"mapTemplateParams" : {
"GIVEN_SPATIAL_LITERAL" : "\"POLYGON((23.708496093749996 37.95719224376526,22.906494140625
     40.659805938378526,11.524658203125002 48.16425348854739,-0.1181030273437499
     51.49506473014367,-3.2189941406250004 55.92766341247031,-5.940856933593749
     54.59116279530599,-3.1668090820312504 51.47967237816337,23.708496093749996
     37.95719224376526))\"^^<http://www.opengis.net/ont/geosparql#wktLiteral>",
"FUNCTION" : "sfIntersects"
},
"mapGraphPrefixes" : { } }
```

Listing 13 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}
```

B GeoRDFBench Execution Script and Logs

This appendix details the differences of the experiment execution script between RDF4J vs RDF4J+ and GraphDB and GraphDB+ variants as well as, part of the experiment Scalability 10K repository generation for RDF4JSUT+ and part of the execution log of Virtuoso against the Scalability 1M workload.

Listing 14 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
...
```

Listing 15 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
...
```

Listing 16 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
```

Listing 17 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...
```

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
```

C GeoRDFBench Experiment Common Execution Logic

This appendix presents the execution logic of the experiment executor component of the GeoRDF-Bench runtime which is shared by all implemented RDF modules.

Algorithm 1 Experiment execution loop.

```
procedure RUN
    querySetIterations ← 0                                  ▷ queryset iteration counter
    problematicQueries.clear()                              ▷ queries that raised exception
    while max experiment duration has not been exceeded do
        queryset initialization                             ▷ dynamic or file-driven
        for all COLD executions do                          ▷ COLD runs
            for all queries in filtered queryset do
                for all repetitions do
                    init SUT after clearing caches
                    coldResult ← sut.runTimedQuery()
                    add coldResult to resCollector,reportSink
                    close SUT
                    if coldResult raised exception then
                        skip remaining query repetitions
                    end if
                end for
            end for
        end for
        for all WARM executions do                          ▷ WARM runs
            for all queries in filtered queryset do
                if COLD run raised exception then
                    warmResult ← coldResult
                    add warmResult to resCollector,reportSink
                    skip to next query
                end if
                init SUT after clearing caches
                pseudorunResult ← sut.runTimedQuery()
                for all repetitions do
                    warmResult ← sut.runTimedQuery()
                    add warmResult to resCollector,reportSink
                    close SUT
                    if warmResult raised exception then
                        skip remaining query repetitions
                    end if
                end for
            end for
        end for
```

Algorithm 1 Experiment execution loop (cont.).

```
        for all COLDCONTINUOUS executions do                    ▷ CONT runs
            if querySetIterations = 0 then                      ▷ init system once
                init SUT after clearing caches
            end if
            for all queries in filtered queryset do
                if query is contained in problematicQueries then
                    skip to next query
                end if
                coldContResult ← sut.runTimedQuery()
                add coldContResult to resCollector,reportSink
                if coldContResult raised exception then
                    add query to problematicQueries
                end if
            end for
        end for
    end while
    close SUT and
    flush report sink deferred records
    resCollector.exportStatistics()
end procedure
```

D Docker Examples Feature Matrix

This appendix presents the features exhibited in each one of the docker examples found in the GeoRDFBench web site [32].

Table 13 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**