The XXXX-2 Framework: Geospatial semantic benchmarking simplified

Anonymous Author(s)

Abstract

We present the XXXX-2 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 geospatial aspects, allow for the automatic mapping of datasets to graphs, provide a specialization hierarchy of queryset types for micro or macro experimental scenarios, even for modeling dynamically generated queries. Queries may define their expected resultset to enable automatic accuracy verification. Experiment behavior and execution logic is controlled by the execution specification, which dictates the action (run experiment or print ground queries) to take, the number of repetitions per execution type (cold, warm, continuous cold), the query repetition and experiment timeouts, the delay period before clearing caches, the aggregating function for reporting execution times, and the policy to follow upon cold execution time out. We decouple these declarative benchmark specifications from the framework's execution engine and serialize them as JSON files; this way, we increase their reuse (instantiation through deserialization), experiment reproducibility and dissemination. We also model the Geospatial RDF store optional application and database server modules and manage their life-cycle (start, stop, restart) during experiment execution to achieve ideal cold cache query executions. In addition, we unify by generalization the repository and connection functionalities of the three most common RDF framework Java APIs offered by RDF stores: OpenRDF Sesame, Eclipse RDF4J and Apache Jena. At the same time XXXX-2 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.

CCS Concepts

• **Information systems → Spatial-temporal systems**; **Semantic web description languages**.

Keywords

geospatial, semantic, benchmarking, framework

SIGMOD '26, Bangalore, India

ACM Reference Format:
Anonymous Author(s). 2025. The XXXX-2 Framework: Geospatial semantic benchmarking simplified. In *Proceedings of ACM on Management of Data (SIGMOD '26).* ACM, New York, NY, USA, 14 pages.

1 Introduction

Many projects [11–13] 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 [35]. Graph database systems with varying degrees of spatial support have recently been used to manage very large LOD datasets [6, 8, 15, 17, 25, 26, 30, 33, 48]. 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 [9, 16, 21, 22, 37] have assisted in creating datasets of desired size and attributes, while log-mining techniques [16, 37, 55] 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 [51]. In all cases however, the benchmark researcher has not been spared the effort to deal with system configuration and optimization, spatial indexing setup, detailed query execution, exception handling, and learning required technology stacks and platform APIs.

Motivation for this work. Our experience with geospatial benchmarks on single node and Spark-based distributed RDF stores along with synthetic data and query generation[1] has led us to believe that the geospatial semantic store research area would greatly benefit by the introduction *of a lean but extensible geospatial benchmarking framework that aims to assist system evaluators in creating new customizable benchmarks with many different systems, many workload types, in as fast, credible and repeatable way as possible.*

Our work, at this stage, focuses on benchmarking single node GeoSPARQL/SPARQL query engines through the console. It makes it easy integrating new or updated RDF stores, provides a benchmark model that 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[2]. Perpendicular to our direction, HOBBIT [51] provides a generalized architecture with message bus connected containers for benchmarking the entire LD life cycle. However this also forces it to be completely agnostic and therefore

[1] Citations omitted due to double blind reviewing.
[2] XXXX-2's site includes containerized images for demonstration purposes.

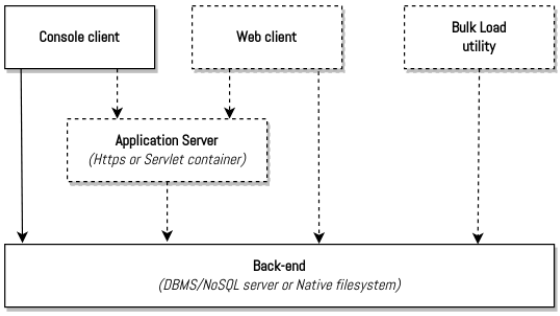
provide no assistance to its users for systems' integration, dealing with the spatial aspects of the benchmark workloads or handling the specific GeoSPARQL/SPARQL query language structure, features and execution issues.

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 [25] benchmark's components, a PostgreSQL and an H2 [38] embedded implementation of the JDBC report sink and hosts with Linux and Windows operating systems*.
(2) The runtime *abstracts and generalizes repository connection functionality from the three best known RDF framework APIs*: (i) OpenRDF Sesame, (ii) Eclipse RDF4J and (iii) Apache Jena. At the same time, *it explicitly models the RDF store application and back-end modules by embedding their life-cycle management in the experiment execution*. It combines the above by *implementing class hierarchies of geospatial enabled RDF stores* according to their basic module architecture and the RDF framework APIs they support.
(3) The framework *provides example implementations for geospatial RDF stores* of different architectures and different RDF framework APIs. These come in the form of Maven modules[3] and the list includes: *(i) RDF4J [49] with or without Lucene SAIL [50], (ii) GraphDB [41], (iii) Stardog [56], (iv) Strabon [33], (v) Virtuoso [17], (vi) Jena GeoSPARQL [45]*. 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*[4] 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.

[3]XXXX-2 framework is a Java Maven multi-module POM project.
[4]Usually due to invalid geometries or unsupported operators.

Figure 1: Geospatial Graph Store Architectures



(5) The framework can also be used for *SPARQL benchmarks*. LUBM [22] benchmark's workload component is included in the JSON specification library.

The organization of the rest of the paper is as follows. Section 2 discusses related work. Section 3 presents the high level architecture of the framework while section 4 explains XXXX-2's runtime. Showcasing the JSON generator API follows in section 5. Section 6 shows an evaluation of the framework and section 7 compares XXXX-2 and HOBBIT. Section 8 presents conclusions and future work. Finally, the appendix features a comparison of XXXX-2 to Geographica 2.

2 Related Work and Background

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

Graph Store Categories. Graph stores in general, follow either the Resource Description Framework (RDF) or the Labeled Property Graph (LPG) approach. RDF as a data model has good expressivity, while featuring a standardized declarative query language SPARQL [60] and a standardized spatial vocabulary GeoSPARQL [36]. LPGs, on the other hand, excel in graph traversal and path search for analytics and machine learning. But, until very recently, they lacked standardization as there are several data models and languages from high-profile vendors and institutions, such as, Neo4j's Cypher [20], Apache TinkerPop Gremlin [52], the Oracle supported PGQL [59] and G-Core [5] from the Linked Data Benchmark Council (LDBC). Another issue is that some of these languages are declarative while others are procedural. As of April 2024, the first edition of the Graph Query Language (GQL) standard [1] is officially published and hopefully will allow language inter-operation with SPARQL. To conclude, at the time of writing of this paper, most geospatial graph databases that support complex geometries support the RDF model.

Geospatial Graph Store Architecture. The available geospatial graph stores feature a 3-tier architecture with standard and optional components. Their architecture is similar to that in Figure 1 which shows their standard and optional components.

All stores have some front-end application, usually a terminal or web-based `console`, through which the user can create repositories, load datasets to and run queries against them. Depending on the

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 [6] uses the embedded key-value database Berkeley DB with a standard R-tree spatial index, Strabon [33] uses PostgreSQL and PostGIS with an R-tree over GiST [23] 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 [44] uses an R-tree spatial index on top of its proprietary industry leading RDBMS solution. More contemporary market-driven stores offer elastic architectures which effectively decouple modules from specific implementation choices by offering many compatible alternatives for each one of them. For example, Virtuoso offers virtual graphs over many well-known data and file formats such as Excel, XML files and RDBMS data sources.

Geospatial Graph Store Evaluation Criteria. The growth rate of LOD sizes questions the ability of graph databases to persist this big data, while at the same time it poses a critical challenge to the performance of these stores under query loads of interest [8, 32]. 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 [19] and geohash [61] 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.

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 [22], BSBM [9], DBpedia SPARQL benchmark (DBPSB) [37] and the Social Network Benchmark (SNB) [16], just to mention a few. GeoSPARQL benchmarks have been presented in [45, 47], the benchmark Geographica in [21], a smart city services related benchmark in [7], a compliance benchmark in [29] and Geographica 2 in [25].

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

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

In particular, several SPARQL and GeoSPARQL benchmarks have generalized or automated the queryset and dataset generation task. For example, LUBM, which focuses on reasoning, features a university ontology-based synthetic data generator able to scale to arbitrary sizes. DBPSB's queryset creation process is based on querylog mining, clustering and SPARQL feature analysis, which is applied to the DBpedia knowledge base and shows that performance of triple stores is by far less homogeneous than suggested by non application-specific benchmarks. FEASIBLE [55] 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 [14] 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

[5]Removing problematic queries or reconfiguring the system.

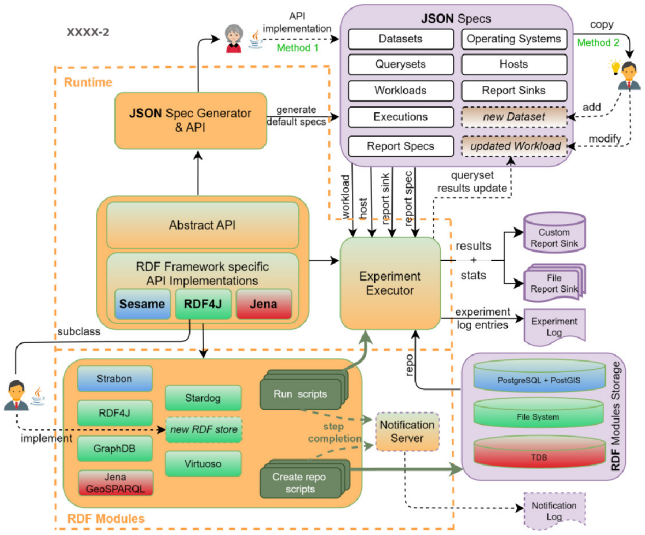


Figure 2: Architectural overview of XXXX-2 Framework

and parallel requests. LITMUS [57] proposes uniform benchmarking of non-spatial data management systems supporting different query languages, such as, SPARQL and Gremlin. Geographica and Geographica 2 include an ontology-based geospatial synthetic generator able to create spatial datasets of arbitrary size and also generate the corresponding queryset with a user defined thematic and spatial selectivity. Kobe [31] cloud benchmarking engine for federated query processors includes the GeoFedBench [58] benchmark, which focuses on validation of the actual crop land usage against the Austrian land survey dataset.

Benchmarking Framework Platforms. These are multi-user environments where researchers can store and share datasets, querysets, execution results and system modules. They are designed for deployment to cloud infrastructures, with distributed file systems and containerization technologies. HOBBIT [51], the most complete of these platforms, extends the scope of benchmarking to the entire linked data life-cycle [40], such as *link discovery* [27] and allows the integration of systems in various programming languages. For a comparison between XXXX-2 and HOBBIT please refer to Section 7.

3 XXXX-2: A Framework Simplifying Geospatial Semantic Benchmarking

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

The system consists of two main parts: the *runtime* and the *RDF modules*, depicted inside the dashed border. The runtime is the engine and fabric of the framework and it is responsible for generating JSON benchmark specifications and executing experiments initiated by the RDF modules' run scripts. The RDF modules is where pre-implemented and newly implemented RDF stores reside that can participate in experiments. Stores use their *repository creation script* to create repositories and import data to them. Each store's

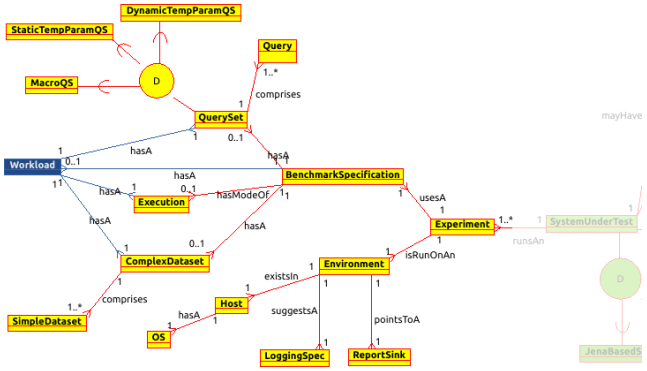


Figure 3: EER diagram of Abstract API - Experiment Components

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

4 XXXX-2 Runtime: The Framework's Engine

The XXXX-2 runtime consists of four components: (i) the `Abstract API`, (ii) the `RDF Framework Specific API Implementations`, (iii) the `Experiment Executor` and (iv) the `JSON Specification Generator`.

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

4.1 Abstract API

This is a core part of the XXXX-2'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[6] 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.

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

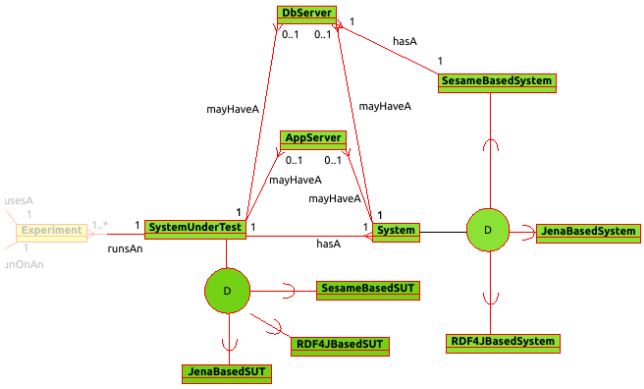


Figure 4: EER diagram of Abstract API - System Under Test

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

4.1.1 Experiment Components. Figure 3 depicts the experiment components, which are detailed below and which are logically organized into the *Benchmark Specification* and *Experiment Environment* groups.

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

Dataset. The entity `ComplexDataset`, in Figure 3, represents the "complex" or "composite" geospatial dataset which can comprise one or more "simple" geospatial datasets represented by the `SimpleDataset` entity. The *simple dataset* specification contains: a logical name, the dataset relative path location, the filename, the RDF serialization format, a map of dataset relevant namespace prefixes, a map of properties that link features with their geometries which represent their spatial extent e.g., `geo:hasGeometry`, a map of properties that link a geometric element with its WKT serialization e.g., `geo:asWKT`, and the scaling factor used in case of a synthetic dataset. The *complex dataset* specification contains: a logical name, the dataset base path location, a map of contexts (named graphs) and the list of simple datasets comprising it. Experiments use only complex datasets.

Queryset. The entity `QuerySet`, in Figure 3, represents the geospatial queryset which can comprise one or more queries represented by the `Query` entity. The *query* specification contains: a label, the GeoSPARQL query text which may contain replaceable tokens (template parameters), a flag that signals the existence of spatial predicate and the accuracy validation indicator. The accuracy indicator denotes whether the expected number of results returned by the query is dataset-dependent, template-dependent, or independent. The *queryset* specification contains: a logical name, the queryset path location, a map of queryset relevant namespace prefixes, a map of fixed, static template, or dynamic template queries with arithmetic index and replacement maps that assist in creating ground queries from template queries. Experiments use only querysets, which *can be inclusively or exclusively filtered at run time*. The `StaticTempParamQS` subclass models sets of template parameter queries which have fixed parameter values for all queries. `DynamicTempParamQS` subclass models sets of template parameter queries which may have different value for each parameter for each query. While these subclasses are useful for "micro"[7] experiment types, the `MacroQS` subclass and its specializations[8] are useful for "macro"[9] experiment types.

Execution specification. The entity `Execution`, in Figure 3, describes which experiment action *(**run**, print)* to take, the query execution types *(cold, warm, cold_continuous)* and number of repetitions per type, the query repetition timeout and total timeout for all repetitions, the delay period for synchronous clearing of system caches, the function to use for aggregating execution times and the policy to follow when a cold execution times out. The non-default *print* action triggers a pseudo-execution which generates the ground queryset for inspection purposes.

***Workload** (compact form).* The entity `Workload`, in Figure 3, represents the compact form of the benchmark experiment description: *dataset + queryset + execution specification*.

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

Operating system. The entity `OS`, in Figure 3, represents the host's operating system and features a name, the shell command path, the commands for synchronizing cached data to persistent storage and the one for fully clearing caches (pagecache, dentries and inode).

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

Report sink. The entity `ReportSink`, in Figure 3, describes the experiment result report store, where the customized reports and statistics will be sent. The default report store is a PostgreSQL JDBC implementation and has as properties, the driver name, hostname, alternate hostname, port, database name, user and password. Alternate hostname allows for having a fall-back database where results from extremely long running experiments can be saved. The PostgreSQL report store has as default behavior the *deferred insertions* for query execution results, that involves an experiment result collector which flushes results upon experiment termination. The target report sink database schema is generated with the help of the runtime-bundled *database generation SQL script*.

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

Logging or Report specification. The entity `LoggingSpec`, in Figure 3, allows customization of the number of resultset entries to be logged during the query execution scanning phase of the `Experiment Executor`. A positive non zero integer value allows for a sample of the results returned by each query to be recorded in the experiment log and can be used as a proof of concept that a system performs accurately or similar to other systems. Such a setting is useful in early benchmarking phases and can help identify, early on, issues with disabled plugins, external libraries, or with incorrect results by non-compliant function behavior. A zero value, on the other hand, allows for very accurate calculation of the query response time and is useful in the final benchmarking phase.

4.1.2 Systems and SUTs. In Figure 4, the parent concepts of system and system under test (SUT) components only, are also part of the Abstract API. The entity *System* represents an RDF framework independent geospatial semantic store and more specifically the repository aspect of it. It is described by a map of properties and their values, such as repository location and name, system relevant namespace prefixes, as well as various indexing parameters. It also has a connection property which allows query execution and a flag to denote whether the store has been initialized. The entity *SystemUnderTest* on the other hand represents the combination of a *System* with its optional application and database server components, represented by *AppServer* and *DbServer* respectively.

On the implementation level, the Abstract API comprises two layers: (i) the *Geospatial System Abstraction Layer*, which is depicted in the lower left two hierarchy levels of Figure 5, and (ii) the *System Under Test (SUT) Abstraction Layer*, which is the lower right level of the same figure, both of which are explained below.

Geospatial System Abstraction Layer. This layer comprises one interface that describes a geospatial RDF store and one abstract class that implements the RDF framework independent common functionality. The *Geospatial Graph System Interface* (`IGeographicaSystem`), is a contract that requires functions for: setting a map of system properties, system initialization, system termination and a function that returns system specific namespace prefix mappings. The *Base Abstract Implementation* (`AbstractGeographicaSystem`) of `IGeographicaSystem`, is an abstract class that uses generics and encapsulates the system properties map, the initialization status, the generic repository "connection", which is RDF Framework specific and the skeleton functionality to handle these. This generic "connection" corresponds to an appropriate RDF Framework abstraction that allows creating query instances on a system repository.

System Under Test (SUT) Abstraction Layer. This layer comprises the generic *Geospatial Graph SUT Interface* (`ISUT`), which is a contract that requires functions for: retrieving the host, the generic "system", execution and report specifications, starting and terminating the application and database server, making system dependent translations of the queryset and executing timed queries.

4.2 RDF Framework Specific APIs

The second part of the core API, on the conceptual level, is depicted in part of Figure 4 which includes the specializations of system and SUT. In a similar manner, system and SUT concepts have three child entities to model the corresponding three RDF framework specific concepts: *RDF4JBasedSystem*, *JenaBasedSystem*, *SesameBasedSystem*, *RDF4JBasedSUT*, *JenaBasedSUT* and *SesameBasedSUT*.

On the implementation level, this part comprises: (i) the *RDF Framework Specific System Layer*, which is depicted by the left side of "RDF Framework Implementation" level of Figure 5, and (ii) the *RDF Framework Specific SUT Layer*, which is the right side of the same level, both of which are explained below.

4.2.1 RDF Framework Specific System Layer. This layer consists of three specializations of the `AbstractGeographicaSystem` class, one for each RDF framework supported by the XXXX-2 runtime. Each class grounds the generic "connection" to the most appropriate interface or class of the RDF framework it implements. Since each framework has more than one major release, which commonly break backward compatibility, the exact version of each RDF framework supported by the runtime was based on its usage by RDF graph stores. The three specialization classes are:

Sesame API (`SesamePostGISBasedGeographicaSystem`). Most scalable RDF solutions based on Sesame, use v2.6.x since support of the RDBMS Sail was deprecated after that. This Sail allowed graph stores to tap, among other things, into geospatial and other capabilities provided by well known DBMSs' such as PostgreSQL with PostGIS. Therefore, this implementation adds: host, port, database name, user and password, to the system properties map and handles them appropriately. The generic "connection" type is replaced with class:

```
org.openrdf.repository.sail.SailRepositoryConnection
```

RDF4J API (`RDF4JBasedGeographicaSystem`). Version 4.x of RDF4J is not supported by all systems, requires Java 11 as the bare minimum, removes initialize methods on Repository, Sail APIs and RepositoryManager and upgrades Lucene libraries from 7.7 to 8.5 affecting disk indexing. Version 3.7.x on the other hand is widely supported by all systems while still offering the required functionality. This implementation focuses on the NativeStore Sail and adds the repository base directory, repository name and indexes used. The generic "connection" type is replaced with interface:

```
org.eclipse.rdf4j.repository.RepositoryConnection
```

Jena API (`JenaBasedGeographicaSystem`). Only Jena GeoSPARQL uses this API, we simply chose the most frequently used Jena version in Maven Central [39] , from the latest stable branch, which was 3.17.x. Jena Tuple Database (TDB) [46] was preferred over TDB2 as the persistent storage option as it did not raise as many issues during development. This implementation adds the repository base directory, repository name and injects the transaction functionality in the system initialization and termination code. The generic "connection" type is replaced with interface:

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

4.2.2 RDF Framework Specific SUT Layer. This layer comprises three base implementations of the `ISUT` interface with corresponding abstract classes: `SesamePostGISBasedSUT`, `RDF4JBasedSUT` and `JenaBasedSUT`. Each class among other things handles the details of initialization and termination of system, application and database server components of the SUT either as a whole or on

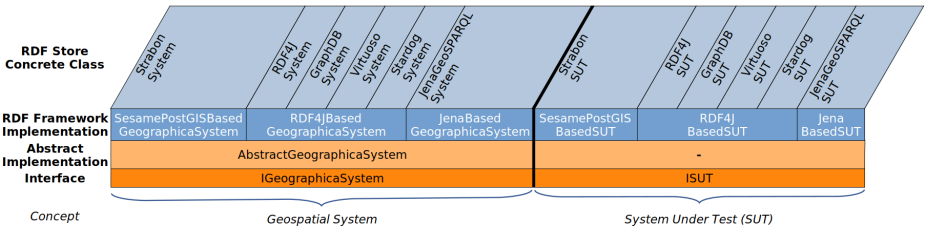


Figure 5: Systems & SUT Class Hierarchies

a component basis. They also invoke system specific query translations, manage and monitor query execution which takes place in a separate thread, such as, enabling timeout for the executing query and handling customized exceptions thrown by different RDF frameworks during the query evaluation phases.

4.3 Experiment Executor

The executor comprises the concrete `Experiment` and abstract `RunSUTExperiment` classes. The subclasses of `RunSUTExperiment` that RDF modules have to implement are the entry points for all experiment run scripts. The `RunSUTExperiment`, parses the script arguments that describe which JSON specifications (see Figure 2) need to be deserialized into experiment component instances, applies queryset filter if needed, configures the SUT with the above and finally launches the `Experiment` run loop. Two actions, performed at the experiment construction time, are the namespace prefix map merging between the corresponding maps of the system, dataset and queryset along with system dependent queryset rewrites in case non standard vocabularies are used offering similar functionality.

4.4 JSON Specification Generator

This runtime component is a collection of runnable utility classes with no parameters, one for each experiment component type, which create all the specifications necessary to run the Geographica 2 benchmark. This geospatial benchmark employs the majority of dataset, queryset and execution specifications' types. These JSON specifications are part of the project build tree and are readily available to the user.

The detailed component type hierarchies, create the need to handle many polymorphic instances which must be properly serialized, otherwise it will be impossible to deserialize them. For this purpose, the Jackson [18] JSON library is used to annotate interfaces and class hierarchies to simplify serialization and deserialization.

5 JSON Generator API - By Example

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

Scalability Workload Description. This workload (see Table 1) is intended to evaluate geospatial RDF store scalability against increasingly larger real-world datasets with many complex geometries. There are 6 workload variants, each comprising one dataset with increasing number of triples (10K, 100K, 1M, 10M, 100M, 500M), a set with either 3 spatial function queries (1 selection and 2 joins) or with 3 equivalent spatial predicate queries[10] 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.

Table 1: Geographica 2 Scalability workloads

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

5.1 Dataset generation

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

Listing 1: Generate Dataset Specification

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

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

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

Listing 2: Serialized Dataset Specification

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

5.2 Queryset generation

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

Listing 3: Generate Queryset Specification

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

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

Listing 4: Serialized Queryset Specification

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

5.3 Execution spec generation

The following code generates the Scalability execution specification object.

Listing 5: Generate Execution Specification

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

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

Listing 6: Serialized Execution Specification

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

In a similar manner, the user can generate the experiment environment specifications: host, operating system, report sink and logging. At this point we should remind that, as it was envisioned by design, the trivial method of copying and modifying an existing specification file with a standard text editor, will probably suffice for many use cases, once a library of JSON specifications is already available. Both methods of generating specifications are depicted in the upper part of Figure 2.

For an example of generating the *compact form* of a benchmark specification, the user can refer to *XXXX-2 Framework Samples*[11] [3] application that is using as test case, the LUBM benchmark for SPARQL.

6 Experimental Evaluation

In this section, we present the process of running some of the Geographica 2 [25] benchmark scalability experiments with the help of the XXXX-2 framework and present the results. The experiment environment, benchmark description and execution details are presented below.

6.1 Environment

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

In addition PostgreSQL v14.7 with PostGIS v3 was also installed, since Strabon requires it for creating spatial databases. The XXXX-2 was uncompressed in `/data/XXXX-2`. The project come ready with prebuilt binaries. To setup the report sink, we run the `scripts/geographica3.sql` that creates the corresponding database:

```
user@NUC8i7BEH:$/data/XXXX-2$ sudo -i -u postgres
postgres@NUC8i7BEH:~$ psql -g /data/XXXX-2/scripts/geographica3.sql
```

Choice of Systems. We chose the following 4 systems to participate in this demonstration: (i) GraphDB, (ii) RDF4J without Lucene SAIL, (iii) Stardog and (iv) Strabon. The first three use the RDF4J Framework while the last one uses OpenRDF Sesame. Stardog and GraphDB also required that we install the corresponding server software and their licenses under `/data`.

6.2 Benchmark description

Since the experiment aims primarily at exhibiting the usage of XXXX-2 for running a benchmark rather than exposing performance issues of the SUTs, we chose the 10K, 100K and 1M variants of the Scalability workload (see Table 1). The following listing shows the ground queryset for RDF4J:

Listing 7: RDF4J Ground Queries & Prefix Header

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

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

[11] Anonymized repo due to double blind reviewing.

XXXX-2, during experiment execution, *automatically constructs the prefix header to include system, queryset and dataset specific namespace prefixes*. For example, while Strabon headers do not differ from the one presented above, the headers for GraphDB and Stardog also include the following system specific namespace prefixes:

Listing 8: GraphDB & Stardog Prefix Header Differences

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

To produce the ground queries of Listing 7 XXXX-2 conveniently takes the necessary steps. First, the template parameters of the queries in Listing 4 are *replaced with their values during queryset deserialization*. After that, it *applies any system provided query translations in order to create the ground queries*. For example, Stardog emulates the GeoSPARQL `geof:sfIntersects` function with the combination of the non-standard `geof:relate` through the negation of `geo:disjoint` as exhibited below for query SC1:

Listing 9: Stardog Queries SC1 Differences

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

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

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

6.3 Repository Creation

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

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

```
user@NUC8i7BEH:/data/XXXX-2/scripts$ source ./prepareRunEnvironment.sh
 NUC8I7BEH RDF4JSUT "CreateRDF4JRepos"
```

6.3.2 Review environment variables. The environment variables can be reviewed with the **print environment script**:

```
user@NUC8i7BEH:/data/XXXX-2/scripts$ ./printRunEnvironment.sh
All SUTs
--------
Environment = NUC8I7BEH
DatasetBaseDir = /data/Geographica2_Datasets
ActiveSUT = RDF4JSUT
...
RDF4J SUT
---------
RDF4JRepoBaseDir = /data/RDF4J_3.7.7_Repos/server
Version = 3.7.7
...
```

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

Listing 10: Modifying RDF4J RepoGen Wrapper Script

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

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

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

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

6.4 Run expirements

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

```
user@NUC8i7BEH:$ export JSONLIB=/data/XXXX-2/json_defs
user@NUC8i7BEH:$/data/XXXX-2/RDF4JSUT/scripts/RunTests3/runTestsForRDF4JSUT.sh
-Xmx24g  # max RAM allocated
-rbd RDF4J_3.7.7_Repos/server # relative base path for RDF4J repositories
-expdesc 172#_2024-02-20_RDF4JSUT_Run_Scal10K # experiment description
-ds ${JSONLIB}/datasets/scalability_10Koriginal.json # dataset
-qs ${JSONLIB}/querysets/scalabilityFuncQSoriginal.json  # queryset
-es ${JSONLIB}/executionspecs/scalabilityESoriginal.json # execution
-h ${JSONLIB}/hosts/nuc8i7behHOSToriginal.json # host
-rs ${JSONLIB}/reportspecs/simplereportspec_original.json # logging
-rpsr ${JSONLIB}/reportsources/ubuntu_vma_tioaRepSrcoriginal.json # report sink
```

Width sanitized sample output is shown below:

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

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

The run script starts by evaluating the arguments and deserializing any experiment related JSON specifications. Then, it performs system specific query translations and namespace prefix header merging. The experiment begins by executing the first iteration of the first query SC1 with COLD caches. The full query to be executed is displayed, the system caches are cleared since a COLD cache execution has been requested and after a few second delay for Java garbage collection, the experiment launches an `RDF4JbasedExecutor` instance in a new thread for executing the query in a time constrained manner. The child process reports back every 6 hours (21600000 msecs) which is the 25% of the total query timeout (24 hours)[12] specified in the scalability execution specification. The executor reports all state transitions for the query execution and during the scanning phase reports the first 3 results of the resultset according to the `simplereportspec_original.json` logging specification. Before exiting, the executor reports that it completed with no errors (COMPLETED-NONE), the query evaluation, scanning and total time in nano seconds, the number of results in the resultset, the number of scan errors[13] and that the accuracy could not be verified, because the queryset specification

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

provided did not include the expected number of results for this query. The collector then flushes in the report sink (PostgreSQL geographica3 database) the `2 cache types * 3 queries * 3 iterations = 18 results` and also generates the standard result and statistics files in the file system.

6.5 Evaluation Results

Table 2 presents the evaluation results for the XXXX-2 Framework. All four systems calculate the correct number of results for all queries. Overall, Strabon performs the best in almost all cache type and query combinations and exhibits a major improvement of warm cache performance over cold cache. Second best is RDF4J which performs predictably since warm cache times are faster than cold ones, response times for higher selectivity join SC3 are smaller than lower selectivity join SC2 which means that filters are considered early in the query plan and finally scales in an analog manner when repository size grows. GraphDB and Stardog share the third position but with different strengths and weaknesses. They share similar performance for SC1 with warm caches. GraphDB performs better in SC1 cold and SC3 and provides marginal improvements with warm caches if selectivity is not low. Stardog performs much better than GraphDB in the low selectivity spatial join SC2 which indicates that its data storage scheme performs better in full scans than GraphDB's storage scheme. Stardog warm cache times show concrete improvements over cold cache ones, but independent of cache type, performance degrades faster as repository size scales. This is consistent with the behavior of RocksDB [10] key-value database of Stardog, whose read overhead gets very high as the number of files (repository size) to be merged grows.

The results of the above benchmark are available through our repository [2].

6.6 Benchmarking with Docker - Examples

The XXXX-2 web site[14] contains detailed tutorials and 4 Docker examples. The *"Multiple stores against multiple Scalability Workloads"* example engages RDF4J, GraphDB and JenaGeoSPARQL against the Scalability 10K, 100K, 1M workloads and exhibits features such as: (i) customization (**Method 2**) of queryset specification, (ii) query accuracy checking and (iii) custom sink reporting. In the *"Multiple stores against LUBM(1,0) Workload"* example XXXX-2 tests RDF4J, GraphDB and JenaGeoSPARQL against the LUBM(1,0) SPARQL benchmark workload and exhibits features such as: (i) generation (**Method 1**) of workload compact specification, (ii) query inclusion filtering, (iii) query accuracy checking and (iv) SPARQL benchmark compatibility. The *"Strabon and Virtuoso against the Synthetic Workload"* example engages Strabon and Virtuoso against the Geographica 2 *Synthetic-512* workload and exhibits features such as: (i) query inclusion filtering, (ii) customization (**Method 2**) of execution specification, (iii) query timeout handling and (iv) custom sink reporting. All the above are shown in more detail in Table 3.

7 Comparison of XXXX-2 and HOBBIT

In this section we compare XXXX-2 against the HOBBIT benchmarking platform to showcase new features and improvements that are introduced by it.

[14]Web site citations omitted due to double blind reviewing.

HOBBIT [51] is a very powerful but equally generic benchmarking framework that *uses linked data technologies but it does not provide any special assistance for benchmarking the performance of Linked Data (LD) loading to and SPARQL query execution against RDF stores*. Also, *no provision for GeoSPARQL is present anywhere in the framework*. XXXX-2, on the other hand, was specifically designed for benchmarking RDF data loading and GeoSPARQL/SPARQL query execution using the Java language.

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

HOBBIT has been used to benchmark non-LD applications such as machine learning (ML) applications [54] and, among other things, allows it to run benchmarks throughout the LD life-cycle [28, 54]. This flexibility is achieved by containerizing the benchmarking components and using the RabbitMQ [34] message broker, which allows *byte array* "messages" to be exchanged between them. This *stripped from semantics representation* is exactly what powers the general applicability of the HOBBIT framework, but *at the same time it carries a heavy price for the component developers who are obliged to take care all the application domain related critical tasks* without assistance from the platform APIs [24].

For the application domain of "benchmarking GeoSPARQL enabled RDF stores" the developer has to acquire knowledge and implement tasks such as: (i) RDF Store back-end choice with options, e.g., RDF4J+Lucene, (ii) setup optimized configurations (sizing memory resources to various components) which may differ for data load and query execution or when big data are involved, e.g., Virtuoso Performance Tuning [43], (iii) choose between normal and bulk data loaders when available (see GraphDB LoadRDF and PreLoad bulk load utilities), (iv) enable/build a geospatial index for a repository upon creation for some systems, e.g., Strabon, and Stardog, while for other systems such as GraphDB [42] after data loading through a special plugin's operations, (v) choosing spatial index method and accuracy, e.g., GraphDB (vi) tagging GeoSPARQL spatial properties so that RDF Stores can perform spatial operations, e.g., RDF4J+Lucene [50], (vii) build namespace prefix header for each queryset usually needs to merge prefixes from the input dataset, the queryset itself and system dependent prefixes that define required/specialized vocabularies (viii) create a declarative mapping between files and contexts when triples from different origin are loaded in separate graphs (see Geographica 2 Real-world workload), (ix) query execution must be granular, able to handle exceptions though out its phases while being able to receive partial resultsets upon timeouts or gracefully skip results as in the case of "bad geometries".

For such demanding tasks XXXX-2 assists the researcher interested in SPARQL and especially GeoSPARQL benchmarking. Table 4, summarizes the features of both systems.

The *HOBBIT Java SDK* [53] provides evidence [15] of the difficulties working with HOBBIT, such as: (i) demands a lot of reading, (ii)

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

Table 2: Scalability Workload Evaluation - Results & Times

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

Table 3: 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**

Table 4: Feature Comparison with HOBBIT

Category	**Feature**	**HOBBIT**	**XXXX-2**
Generic Features	Programming Language support	**Java, Python,...**	**Java**
	Distributed System support	**yes**	**no**
	SUT Query Language support	**don't care** *(no support)*	**GeoSPARQL, SPARQL**
	Functionality (code) Reusability	**yes***(containers)*	**yes***(Runtime APIs, Scripts)*
	Structure (data) Reusability	**yes***(ontologies)*	**yes***(JSON specs, Scripts)*
	Metadata Model	**ontology**	**ER diagram**
Assistance Testing RDF Stores -SPARQL/ GeoSPARQL Processors	SUT Configuration optimization	**no**	**yes**
	Spatial Index setup	**no**	**yes**
	Query Namespace Header management	**no**	**yes***(auto)*
	Dynamic Queryset filtering	**no**	**yes***(Inc/Exclusive)*
	Dataset To Context mapping	**no**	**yes**
	WARM Cache Query exec	**yes**	**yes**
	COLD Cache Query exec	**no**	**yes**
	Query Timeout handling	**no**	**yes**
	Query Exception handling	**no**	**yes**
	RDF Framework Providers	-	**OpenRDF Sesame, RDF4J, Apache Jena**

requires manual and error-prone work, (iii) does not allow to debug locally, etc. The HOBBIT Java SDK attempts to alleviate some of the problems.

8 Conclusions and Future Work

We presented the concepts, architecture and basic operation of the XXXX-2 Framework, which aims to: (i) save the geospatial semantic benchmark researcher's time and effort testing new systems, (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[16] [3, 4] and site[17].

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

References

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

[2] Author1 A. 2025. *XXXX-2 Framework Benchmark Results on Anonymous Github.* https://anonymous.4open.science/r/XXXX-2-B1C4/

[3] Author1 A. 2025. *XXXX-2 Framework Samples source code on Anonymous Github.* https://anonymous.4open.science/r/XXXX-2_samples-B681/

[4] Author1 A. 2025. *XXXX-2 Framework source code on Anonymous Github.* https://anonymous.4open.science/r/bench-38E0/

[16] Anonymized repos due to double blind reviewing.

[17] Web site citation omitted due to double blind reviewing.

[5] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1421–1432. doi:10.1145/3183713.3190654

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

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

[8] Dimitris Bilidas, Theofilos Ioannidis, Nikos Mamoulis, and Manolis Koubarakis. 2022. Strabo 2: Distributed Management of Massive Geospatial RDF Datasets. In *The Semantic Web–ISWC 2022: 21st International Semantic Web Conference, Virtual Event, October 23–27, 2022, Proceedings*. Springer, 411–427.

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

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

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

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

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

[14] Felix Conrads, Jens Lehmann, Muhammad Saleem, Mohamed Morsey, and Axel-Cyrille Ngonga Ngomo. 2017. I guana: a generic framework for benchmarking the read-write performance of triple stores. In *The Semantic Web–ISWC 2017: 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II 16*. Springer, 48–65.

[15] Alishiba Dsouza, Nicolas Tempelmeier, Ran Yu, Simon Gottschalk, and Elena Demidova. 2021. WorldKG: A World-Scale Geographic Knowledge Graph. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*. ACM, 4475–4484.

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

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

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

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

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

[21] George Garbis, Kostis Kyzirakos, and Manolis Koubarakis. 2013. Geographica: A benchmark for geospatial rdf stores (long version). In *The Semantic Web–ISWC 2013: 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II 12*. Springer, 343–359.

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

[23] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. Generalized Search Trees for Database Systems. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio (Eds.). Morgan Kaufmann, 562–573. http://www.vldb.org/conf/1995/P562.PDF

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

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

[26] Krzysztof Janowicz, Pascal Hitzler, Wenwen Li, Dean Rehberger, Mark Schildhauer, Rui Zhu, Cogan Shimizu, Colby K. Fisher, Ling Cai, Gengchen Mai, Joseph Zalewski, Lu Zhou, Shirly Stephen, Seila Gonzalez Estrecha, Bryce D. Mecum, Anna Lopez-Carr, Andrew Schroeder, Dave Smith, Dawn J. Wright, Sizhe Wang, Yuanyuan Tian, Zilong Liu, Meilin Shi, Anthony D'Onofrio, Zhining Gu, and Kitty Currier. 2022. Know, Know Where, Knowwheregraph: A Densely Connected, Cross-Domain Knowledge Graph and Geo-Enrichment Service Stack for Applications in Environmental Intelligence. *AI Mag.* 43, 1 (2022), 30–39. doi:10.1609/aimag.v43i1.19120

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

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

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

[30] Nikolaos Karalis, Georgios M. Mandilaras, and Manolis Koubarakis. 2019. Extending the YAGO2 Knowledge Graph with Precise Geospatial Knowledge. In *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11779)*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon (Eds.). Springer, 181–197. doi:10.1007/978-3-030-30796-7_12

[31] Charalampos Kostopoulos, Giannis Mouchakis, Antonis Troumpoukis, Nefeli Prokopaki-Kostopoulou, Angelos Charalambidis, and Stasinos Konstantopoulos. 2021. KOBE: Cloud-native Open Benchmarking Engine for federated query processors. In *The Semantic Web: 18th International Conference, ESWC 2021, Virtual Event, June 6–10, 2021, Proceedings 18*. Springer, 664–679.

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

[33] Kostis Kyzirakos, Manos Karpathiotakis, and Manolis Koubarakis. 2012. Strabon: A semantic geospatial DBMS. In *The Semantic Web–ISWC 2012: 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I 11*. Springer Berlin Heidelberg, 295–311.

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

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

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

[37] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. 2011. DBpedia SPARQL benchmark–performance assessment with real queries on real data. In *The Semantic Web–ISWC 2011: 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I 10*. Springer, 454–469.

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

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

[40] Axel-Cyrille Ngonga Ngomo, Sören Auer, Jens Lehmann, and Amrapali Zaveri. 2014. Introduction to linked data and its lifecycle on the web. *Reasoning Web. Reasoning on the Web in the Big Data Era: 10th International Summer School 2014, Athens, Greece, September 8-13, 2014. Proceedings 10* (2014), 1–99.

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

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

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

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

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

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

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

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

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

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

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

[52] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. ACM. doi:10.1145/2815072.2815073
[53] Michael Röder. 2018. *HOBBIT Java SDK web site.* https://github.com/hobbit-project/java-sdk
[54] Michael Röder. 2024. *HOBBIT Examples source code on Github.* https://github.com/hobbit-project/hobbit.examples
[55] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. Feasible: A feature-based sparql benchmark generation framework. In *The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I 14*. Springer, 52–69.
[56] Stardog. 2024. *Stardog.* https://www.stardog.com/
[57] Harsh Thakkar. 2017. Towards an open extensible framework for empirical benchmarking of data management solutions: LITMUS. In *The Semantic Web: 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28–June 1, 2017, Proceedings, Part II 14*. Springer, 256–266.
[58] Antonis Troumpoukis, Stasinos Konstantopoulos, Giannis Mouchakis, Nefeli Prokopaki-Kostopoulou, Claudia Paris, Lorenzo Bruzzone, Despina-Athanasia Pantazi, and Manolis Koubarakis. 2020. GeoFedBench: A Benchmark for Federated GeoSPARQL Query Processors.. In *ISWC (Demos/Industry)*. 228–232.
[59] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, Peter A. Boncz and Josep Lluís Larriba-Pey (Eds.). ACM, 7. doi:10.1145/2960414.2960421
[60] W3C. 2013. *SPARQL 1.1 Query Language web site.* W3C. https://www.w3.org/TR/sparql11-query/
[61] Wikipedia. 2008. *Geohash wiki page.* https://en.wikipedia.org/wiki/Geohash