Semantic Web 0 (0) 1
IOS Press

GeoSPARQL-Jena: Implementation and Benchmarking of a GeoSPARQL Graphstore

Gregory L. Albiston [a,*], Taha Osman [a] and Haozhe Chen [a]
[a] *School of Computing and Technology, Nottingham Trent University, Nottingham, United Kingdom*
E-mails: gregory.albiston@ntu.ac.uk, taha.osamn@ntu.ac.uk, haozhe.chen2014@my.ntu.ac.uk

Abstract. This work presents a RDF graphstore implementation for all six modules of the GeoSPARQL standard using the Apache Jena Semantic Web library. Previous implementations have provided only partial coverage of the GeoSPARQL standard. There is discussion of the design and development of on-demand indexes to improve query performance without incurring lengthy data preparation delays. A supporting benchmarking framework is also discussed for the evaluation of any SPARQL compliant queries with interfaces provided for integrating additional test systems. This benchmarking framework is utilised to examine the performance of the implementation against two existing GeoSPARQL systems using the Geographica benchmark. It is found that the implementation achieves comparable or faster query responses than the alternative systems while also providing much faster dataset loading and initialisation durations.

Keywords: GeoSPARQL, SPARQL, RDF, Apache Jena, geospatial, geospatial data, geospatial query language, geospatial index

1. Introduction

This work discusses the implementation of a geospatial graphstore complying with the OGC GeoSPARQL standard [1]. Previous works have partially implemented the GeoSPARQL standard and have generally extended relational databases rather than utilising a RDF graphstore. This work describes the design features of the implementation, investigation of parameter tuning and the development of a benchmarking framework for comparison with other GeoSPARQL implementations.

The increasing usage of location-aware devices, e.g. smartphones, Internet-of-Things devices and in-vehicle navigation, has led to an increasing range of applications and datasets reliant upon geospatial data. The interpretation of geospatial data and relations is an established field of study upon which the GeoSPARQL incorporates Semantic Web concepts and principles. The Semantic Web is intended to increase the access and sharing of data through the internet and presents the opportunity to enrich data through knowledge modelling, automated inferencing and interconnection of datasets to provide deeper insights and experiences.

A key technology of the Semantic Web is the Resource Description Framework (RDF) [2] which is a data structure standard based upon a directed labelled graph of subject-predicate-object triples. It is intended that any data can be represented using RDF's design principles and extended by vocabularies, e.g. GeoSPARQL, to allow consistent interpretation across datasets. The SPARQL query language [3] provides a query mechanism to explore, retrieve and modify RDF data. It also includes many operations commonly found in the Server Query Language (SQL) used in traditional relational databases. The GeoSPARQL standard enhances SPARQL's query mechanisms to allow users to conveniently access geospatial data.

1.1. Challenges

There are several challenges relating to geospatial graphstores. There is no implementation of the GeoSPARQL standard stating full compliance across all six components. In addition, there is no benchmarking framework to investigate GeoSPARQL compliance and consider user quality of life aspects, such as mixed

*Corresponding author. E-mail: gregory.albiston@ntu.ac.uk.

1570-0844/0-1900/$35.00 © 0 – IOS Press and the authors. All rights reserved

coordinate reference system datasets, inferring geometry metadata and converting datasets between geospatial datatypes and coordinate reference systems.

Each of the reviewed implementations uses modified relational databases for persistent storage rather than an RDF graphstore. These relational databases have been adapted for RDF usage rather than developed specifically for the purpose. Therefore, there is potential for graphstore implementations to have design optimisations which improve performance. The extension of relational databases can also present a setup and environment configuration burden that is justifiable in production scenarios but limiting for prototype development and research.

There is also a design challenge between the SPARQL query execution process and the dimensionality of the problem space defined by the GeoSPARQL standard. In SPARQL, query execution is performed on discrete cases of parameters which are represented in string literal form. In the case of geospatial calculations, information is extracted from the string literals, processed and then discarded for the next iteration of query execution. However, the next iteration step may contain one or more parameters used in the previous iteration. Later query iterations may also use the same parameters as a previous iteration after following a different route through the data graph. Therefore, computation is wasted repeatedly extracting information across multiple iterations.

Potential solutions to this are storing the results of previous computations or pre-computation of all potential values. The GeoSPARQL standard defines many functions but a central component is the three sets of *relation families*. These relation families each describe eight spatial relationships between geometry pairs. The relationship between two geometries is invariant and so suitable to computation and storing during data loading. This would change the extraction and processing described previously to a retrieval process. However, the dimensionality of these calculations is excessive. Calculating all the relationships of a very modest 10,000 geometry dataset would require 2.4 billion triples.

This represents an exceedingly large supporting set of data for a small geospatial dataset and introduces issues of pre-computation cost, storage and search durations. Particularly when only a small subset of geometries and relations may be utilised by a user at any time and the new or modified geometries will require recomputation. Graphstores have been developed to handle trillions of triples [4, 5] and so geometry datasets could be much larger than the scale described. Therefore, a compromise is required between pre-computing or retaining related information versus the potential scale and dimensionality of datasets.

1.2. Contributions

The principle contributions of this work can be summarised as:

1. Full GeoSPARQL implementation of all six components, including automatic inferring geometry properties and query re-writing, using an RDF graphstore.
2. Benchmarking framework for GeoSPARQL and SPARQL queries with results and comparison of three GeoSPARQL systems.
3. Discussion and implementation of novel on-demand indexes to improve GeoSPARQL query performance.
4. Identification of areas for review and revision in future versions of the GeoSPARQL standard.

1.3. Related Work

This section some of the work already undertaken in the development and implementation of GeoSPARQL. The GeoSPARQL standard is influenced by the Simple Features standard. This outlines spatial functions for use in SQL relational databases with geometry shapes represented in Well Known Text (WKT) and Well Known Binary (WKB) formats. Simple Features defines a set of geometry relations which are included in the GeoSPARQL standard as one of three *relation families*. Other functions from Simple Features have also been incorporated but with broader usage in the SPARQL context. Therefore, there is a straightforward route for SQL dependent implementations to achieve some GeoSPARQL compliance by adapting existing Simple Features functionality, but additional aspects need to be taken into consideration.

There are numerous implementations of RDF frameworks with many featuring geospatial and sometimes more specifically GeoSPARQL support. These RDF frameworks provide persistent storage through RDF graphstores or relational databases adapted for RDF with a geospatial extension, e.g. PostgreSQL with PostGIS. While these frameworks refer to supporting GeoSPARQL it has not been possible to find any explicit statements of the implemented GeoSPARQL extensions and requirements compliance. As far as we

are aware, there has not been any systematic research or compliance testing of the various RDF frameworks in their support for GeoSPARQL.

The following gives a brief overview to these frameworks to illustrate the fragmented nature of GeoSPARQL support and different implementation approaches. AllegroGraph is a graph database that supports geospatial operations but takes a more generalised approach than the Simple Features or GeoSPARQL standards, with a set of functions focused upon points within a radius or polygons. Apache Jena is a Java based Semantic Web and Linked Data framework that provides various tools for developing applications including persistent storage, SPARQL query engine, and inferencing. It has a spatial module that supports a small set of functions which are unrelated to the Simple Features or GeoSPARQL standards. Apache Marmotta [6] is a Linked Data platform which uses PostgreSQL with PostGIS backend and provides functions from the *Geometry Extension* and *Geometry Topology Extension*, but geospatial support has not been included in any public release due to "portability issues".

Stardog is an RDF data unification platform that supports a subset of non-topological GeoSPARQL functions along with their own geospatial functions. Virtuoso is SQL database with RDF and SPARQL extensions that includes some geospatial support related to the Simple Features standard but not explicitly supporting GeoSPARQL. Parliament integrates the Apache Jena framework with a Berkley DB persistent storage to provide temporal and GeoSPARQL compliant spatial support. However, the Parliament is dependent on obsolete components of Apache Jena and has not been updated to the latest major version. Strabon [7] is a spatiotemporal store that uses the RDF4J framework backed by PostgreSQL with PostGIS to support the *Core*, *Geometry Extension* and *Geometry Topology Extension* components of the GeoSPARQL and its own stSPARQL syntax.

Parliament, Strabon and uSeekM (now unavailable) were examined in the Geographica benchmark [8], which was developed by the same project team as Strabon. In this work, the three systems were benchmarked for data loading durations, 28 individual queries and three sets of linked queries. Strabon generally outperformed Parliament and uSeekM. The query syntax used predated the finalisation of the GeoSPARQL standard and in some cases was for stSPARQL functionality. The datasets and queries have been published with some updates made to comply with GeoSPARQL.

The remainder of this work is organised as follows. In Section 2 is an overview of Spatial Analysis and key topics in the area along with a motivational scenario for its usage. Section 3 discusses the GeoSPARQL standard and the six extension modules. Section 4 describes the implementation design of the GeoSPARQL extension to the Semantic Web library Apache Jena and features to improve query performance. It also details the design and development of the Benchmarking Framework used to evaluate the implementation. Section 5 discusses the challenges encountered during the implementation process. In Section 6 is an evaluation of results obtained when comparing the implementation described in Section 4 with two existing GeoSPARQL systems, Parliament and Strabon, using the developed benchmarking framework. The final section provides a summary of the findings of this work and identifies areas for future work.

2. Overview of Spatial Analysis

This section provides an overview of key components and concepts for spatial analysis. These components provide the theoretical and practical foundations upon which the Simple Features and GeoSPARQL standards have been developed. A supporting illustrative use case is also provided to highlight the practical need for geospatial queries and support.

2.1. Topological models for spatial analysis

A key feature of Geographic Information Systems (GIS) is spatial analysis and querying using topological models of the relationships between spatial objects [9]. Several models have been developed to interpret these relationships between different geometry shapes, including Egenhofer, Region Connection Calculus (RCC) and the Simple Features which are utilised by the GeoSPARQL standard.

Research has been undertaken into different interpretations of spatial information. These have included relationships based on direction and distance using logical operators and comparing spatial data using algebraic point sets. The approach of the Egenhofer Topological Model [9] seeks to overcome limitations of these two approaches by establishing relationships based on the intersection of two spatial objects in their interior and boundary. In this topological model the relationships are invariant to topological transformations, e.g. translation, scaling and rotation, of the spa-

tial objects, focused upon two-dimensional spatial objects and produces a Boolean result. The Egenhofer relation family consists of eight relationships.

The RCC model is a boundary-free theory that seeks to provide a qualitative representation for reasoning based upon axioms that every spatial object is a region, i.e. has an interior, a region connects with itself and if a region connects with another region then the other region also connected to the region. The RCC-8 relation family based upon the assumption of *jointly exhaustive and pairwise disjoint* (JEPD) relations derives eight relationships [10]. The Simple Features relation family is established by the Simple Features standard [11] to describe a set of spatial objects and their relations which satisfy most real-world geospatial data use cases and re-uses relations defined by the previous relation families to form eight relationships.

The Simple Features relation family is represented by the Dimensionally Extended Nine-Intersection Model (DE-9IM). The DE-9IM notation [12] extends the concepts introduced by Egenhofer to describe relations through the intersections of the interior, boundary and exterior of spatial objects. The nine pair-wise intersections produce a matrix where the maximum permitted spatial dimension for the intersection is indicated. The spatial dimension of geometric objects is point 0, line string 1 and polygon 2. The boundary of a geometric object is formed from a set of geometric objects of the next lower dimension. The interior is those points remaining when the boundary is removed, and the exterior is those points not in the interior or boundary. This notation can represent all the relations of the *relation families* while also allowing for alternative relations to be described.

2.2. Spatial Reference Systems (SRS)

The consistent interpretation of spatial coordinate information is managed by Spatial Reference Systems (SRS), also known as Coordinate Reference Systems (CRS) with the terms being used interchangeably here. A wide range of SRS types have been developed for different applications and only key points will be summarised. SRSs can be categorized into several different types with *projected* and *geographic* being two widely encountered. Each SRS contains a coordinate system that describes the relationship between the coordinates and the earth's surface according to a unit of measurement. Coordinate systems can range in dimension or axis number but two or three are common. Two SRS may use the same coordinate system but order the axis differently, e.g. the widely used WGS84 and the GeoSPARQL defined CRS84. Mathematical transformations between SRS have been published as part of SRS definition catalogues, such as EPSG [13].

A *geographic* SRS, e.g. WGS84, places the coordinates on an ellipsoid approximating the earth's shape. This allows the entire earth's surface to be represented using latitude and longitude coordinates in degree or radian units. Accurate distances between points requires consideration of the ellipsoid curvature through *great-circle* or *orthodromic* distance. A *projected* SRS, e.g. Universal Transverse Mercator or UTM, places easting/westing/x and northing/southing/y coordinates on a planar surface. The distortion of representing the earth's curvature on a plane is controlled and compensated with the SRS only being applicable to a small surface area. This area is often enough to represent a whole nation, e.g. United Kingdom has a single *projected* SRS OSGB36/BNG, while the global UTM system divides the earth into sixty SRS zones. This approach allows quick calculation of accurate distances using *Euclidean distance* in linear units, such as metres.

The importance of SRS support can be highlighted by considering the use case of a public SPARQL endpoint. In this case an endpoint is providing access to a dataset of geospatial data encoded in the widely used global *geographic* WGS84. A user is querying the endpoint with a local geospatial dataset encoded in their national *projected* SRS, as is typically the case. A geospatial system that does not support SRS transformations would require the user, or the data publisher, to transform the dataset or all queries into WGS84. This places the technical burden upon on the user and creates the potential for incorrect transformation. It also assumes that the user is aware that such transformation is required or is not supported.

2.3. Spatial Geometry Serialisations

The representation of geospatial data can be achieved using a variety of spatial serialisations, e.g. WKT, GML, GeoJSON and KML. These identify the geometry shape, coordinates and, in some cases, the SRS of the geospatial data. The standards and scope of serialisations vary through application context, defining additional geospatial concepts, such as Features and Geometry, or following the design conventions of a parent serialisation, such as XML and JSON. These serialisations therefore encode equivalent geospatial information, but some variations occur, such as in ge-

ometry shapes and their semantics. Typical geometry shapes are Point, LineString and Polygon along with collections of these shapes.

2.4. Generating travel demand data for traffic simulation: a motivating scenario

An example use case for geospatial data can be found from the domain of Traffic and Transport demand modelling and simulation [14]. This domain investigates the interaction of people, vehicles, locations and transport infrastructure. These interactions can be tangible between physical entities, such as a person's location being upon a pavement area, or abstract, such as a property being in an administrative area.

The undertaking of demand modelling and simulation is multi-stage process with spatial analysis being of use in preparing scenario data, modelling the behaviour of individuals in the scenario's context and analysing the outcomes of simulation. The type of interactions between the three main geometry shapes can be found across each of the stages. The following list provides some example spatial analysis applications that may be encountered:

- Distance between a person (point) and a bus stop (point), road (line string), access/entrance (point) or building (polygon).
- Identifying all the houses (polygon) within a local authority, retail shopping or school enrolment area (polygon).
- Determining the common routes (line string) and stopping locations (point) of individuals during their journeys.
- Identifying access permissions into resident (point) only areas (polygon) during route planning.
- Aligning national demography and transport data (projected SRS) with road and building infrastructure (geocentric SRS).

By applying spatial analysis, the results of these interactions can be consistently and accurately determined allowing greater comparison between models.

3. Extensions and Requirements of the GeoSPARQL Standard

The GeoSPARQL standard was developed to enable querying of geospatial data in RDF format and is structured around six modules supported by thirty conformance requirement clauses. The standard is developed from the Simple Feature standard, which is widely used in SQL databases, but provides a wider definition of spatial relations and incorporates RDF concepts, such as Unique Resource Identifiers (URIs), namespaces and RDFS inferencing.

The Simple Features standard is based upon a simplified spatial model where all coordinates are on a planar surface regardless of SRS. This simplification reduces calculation complexity but introduces potential error when using *geographic* SRS which is tolerated by the standard. Comparison between geometries with different SRS requires accounting for both numerical and positional differences.

3.1. Core

This extension defines the fundamental vocabulary of the standard, which is the *Spatial Object* and *Feature* classes. A *Spatial Object* is the superclass of *Feature* and *Geometry*. A *Feature*, e.g. a building or lake, is defined separately from the *Geometry* that defines its spatial shape. Therefore, a *Feature* may have multiple *Geometry* for different purposes, timestamps or serialisations.

3.2. Geometry Extension

In this module is the definition of the *Geometry* class, the *Geometry Literal* datatype and the *Non-topological Query Functions*. The relationship between *Feature* and multiple *Geometry* is defined using two properties, the one-to-one *hasDefaultGeometry* and the one-to-many *hasGeometry*. The *hasDefaultGeometry* is used in later extensions to select a consistent *Geometry* when determining inferred results.

The spatial coordinates of a *Geometry* are defined between it and a *Geometry Literal* through the one-to-one *hasSerialization* property. The *hasSerialization* property can be sub-classed to a specific serialisation, e.g. *asWKT* or *asGML*, to allow finer control during querying. Additional *Geometry* properties describe information that can be gained by interpreting the serialisation of the *Geometry* but are made available for direct use in SPARQL querying.

The *Geometry Literal* datatype provides the string representation of the geospatial geometry shape and defines two specific serialisations, *WKT* and *GML*, with specific conformance requirements for each. The *Non-topological Query Functions* are applied to one or more *Geometry Literals* to return a result and include distance, intersection and boundary. Calculations are performed in the SRS of the first *Geometry Literal*.

3.3. Topology Vocabulary Extension

This module defines the vocabulary for property relationships, such as *contains*, *equals* and *disjoint*, between two *Spatial Objects* for assertion in a dataset. These relations are defined in three relation family of Simple Features, Egenhofer and Region Connection Calculus 8 (RCC8) with eight *spatial relations* in each family. Each *spatial relation* is defined using a DE-9IM intersection matrix which shows the maximum number of dimensions of the intersection between interior, boundary and exterior of the two geometries.

3.4. Geometry Topology Extension

In this extension the *Topology Vocabulary* is reformulated as filter functions which accept *Geometry Literals* and return a Boolean result. This allows the calculation of *spatial relations* which have not been explicitly asserted in the dataset. Functions are defined for the same three relation family contained in the *Topology Vocabulary*. Calculations are performed in the SRS of the first *Geometry Literal*.

3.5. RDFS Entailment

This extension outlines the application of RDFS and OWL reasoners and geometry hierarchies to produce inferred statements in a dataset. This means that classes and properties, such as *Spatial Object* or *hasSerialization*, can be obtained from a schema applied to the dataset or the stated relationships rather than needing the dataset to consistently state their existence. This reduces the burden upon dataset publishers and allows the automated identification of inconsistencies.

3.6. Query Rewrite Extension

This extension builds upon the vocabulary, functions, and inferencing of the previous extensions to provide the most wide-reaching support to the user. The available usage of a vocabulary for relations between *Spatial Objects* in SPARQL queries, provided by the *Topology Vocabulary*, still requires the dataset to be populated with triples using the vocabulary. This requires the dataset publisher or user to determine and assert the relations. Determining these relations is the functionality of the *Geometry Topology Extension* described previously but is based upon retrieving the *Geometry Literal* of *Spatial Objects*.

This complicates query writing and requires the user to have detailed knowledge of the vocabulary. The *Query Rewrite Extension* allows the user to use the higher level *Spatial Objects* in their queries through syntactic sugar to obtain results regardless of their assertion in the dataset. The geospatial system expands the query as required to find either asserted triples or calculates the relations when absent using the *hasDefaultGeometry* property between *Feature* and *Geometry* to find *Geometry Literals*. This leads to four potential cases of *Feature-Feature*, *Feature-Geometry*, *Geometry-Feature* and *Feature-Feature* and any asserted triples for each *Geometry Literal* pair.

4. Implementation of the GeoSPARQL Jena and Benchmarking Framework

This section describes the design and implementation considerations of the GeoSPARQL Jena system and the supporting benchmarking framework. In the first part is the GeoSPARQL Jena system, which implements the GeoSPARQL standard and extends the Semantic Web library Apache Jena. In the second part is the benchmarking framework, which has been currently been developed for testing three query sets across three target systems and provides facility for interfacing additional target systems and user defined queries.

4.1. Implementing the GeoSPARQL Standard as an extension of Apache Jena

The primary focus of implementing the GeoSPARQL standard is to extend the query execution mechanism of Apache Jena to support all the extension modules. Filter and Property functions are registered with the ARQ query engine for use when matching function or property names are encountered within a SPARQL query. The key implementation difference between Filter and Property functions is that Filter functions only operate upon the parameters passed into the function while Property functions can access the underlying graph to retrieve or create additional triples.

Two design principles were to minimise user configuration requirements and minimise pre-computation through on-demand processing of queried data. This on-demand processing means that initialisation periods are short, only calculations relevant to utilised functionality are performed and new datasets can be

incorporated quickly and with consistent query durations.

The extension approach allows all the existing functionality of Apache Jena, which closely complies with Semantic Web standards, to be utilised, e.g. RDF file handling, RDFS inferencing, HTTP endpoint and persistent storage. The GeoSPARQL class and property hierarchy are achieved by loading the published GeoSPARQL RDF schema into an RDFS inferencing model and applying to any dataset. This allows user and version modifications to the schema to be made without requiring alteration to the implementation source code. The deployment only requires an application developer to write a single configuration line of Java code. Various configuration options for the use of the GeoSPARQL Jena extension have been made available to the developer, such as index sizing and in-memory or persistent storage options. Additional utility methods have also been implemented using the available functionality, such as converting RDF files between spatial reference systems and geometry serialisations.

The implementation workflow, shown in Figure 1, is activated when the ARQ query engine reaches a registered Filter or Property function name. The following describes the workflow with relevant diagram points indicated by parenthesis. The ARQ engine processes query by checking triples in the dataset and passing values to the corresponding function (1). A key concept of SPARQL querying is the reduction of the full dataset graph down to a subset which are true for the graph described by the query. The ARQ engine seeks to optimise execution for the quickest resolution. Therefore, a set of partial results for the wider query may have been obtained prior to passing to the function. Although the function may only be defined once in a query it is called multiple times, once for each potential result, passing in different arguments. There may also be additional parsing of literal values during query execution that is hidden during query writing.

GeoSPARQL Property Functions may retrieve additional values from the dataset (2). Many Property Functions in GeoSPARQL are syntactic sugar for a Filter Function which is used to perform the actual processing. The distinct Property Functions follow the same process as Filter Functions. The Query Re-Write Property Functions also checks for asserted Feature and Geometry relationships in the dataset to shortcut resolution, as required by the GeoSPARQL standard, followed by an additional check of the Query Re-Write Index (2a).

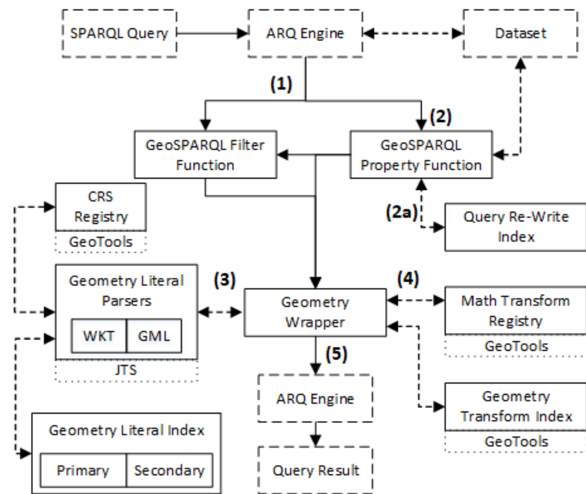

Fig. 1. The GeoSPARQL Jena query execution workflow from SPARQL query to result with key points from the description indicated by parenthesis. Workflow activated when a GeoSPARQL function or property is encountered. Implementation stages shown along with interfacing Apache Jena (dashed) and external library (dotted) stages.

This index contains up to a maximum number of recently found relations between Geometry Literals. Each Geometry Literal pair could be involved in up-to four Feature and Geometry relationships for each spatial relation, without considering that a single Geometry Literal could be used by multiple Features or Geometries. Therefore, retaining recent results has been designed into the workflow to address the re-working behaviour of SPARQL query execution highlighted previously but still respecting the large dimensionality that could quickly result from retaining every result.

Geometry Literals are unparsed into Geometry Wrappers via relevant parsers (3). The Geometry Literal Index is checked for recently unparsed Geometry Literals to allow re-use between iterations through on-demand caching. Geometry Literals are immutable and so later extraction yields the same Geometry Wrapper. The Geometry Literal Index contains two indexes that relate to positions of function arguments which is likely to be consistent between query iterations and allows prioritising the index search. If the Geometry Literal is not present in the initial index, then the alternative index is checked before a new Geometry Wrapper is extracted from the Geometry Literal and added to the initial index. The CRS Registry similarly retains recently used CRS data for any future Geometry Literals with the same CRS.

The Geometry Wrapper handles all processing to resolve the request, e.g. spatial relations and functions,

with one instance for each Geometry Literal (4). The spatial relations and functions leverage the JTS library [15], which is an implementation of the Simple Features standard, and has been extended to incorporate the additional concepts of the GeoSPARQL standard. Mathematical conversions between CRSs are obtained as they are required and stored in a Math Transform Registry for re-use, as the same transformation converts any geometry between two CRSs. The geometry resulting from a CRS conversion is retained by the Geometry Transform Index for future re-use, up-to a configurable maximum number. Here again the design approach has been to retain information that may be useful in later query iterations. The functionality for CRS conversion is provided by the GeoTools library [16] which accesses coordinate transformations definitions provided by defining authorities. The outcome of the function is returned to the ARQ Engine (5) via the calling function to continue processing the query, potentially returning to step (1) for another function.

The retention of data in-memory can present issues when processing large datasets. To manage the sizes of the indexes and registries several strategies have been implemented. Firstly, the data retained within registries is common across multiple geometries and highly re-usable. Therefore, these have been not constrained in size or persistence and are intended to persist for the duration of the application. Secondly, to manage size the index contents expire after a fixed time-period (default: 5 seconds) which is refreshed whenever an item is retrieved. Checking for removals occurs in 1 second intervals. Therefore, memory usage will increase during periods of geospatial querying and then fall back during other activities.

The maximum number of items can also be limited to avoid excessive amounts of memory being used and exceeding available resources. When the index is full, query execution continues unblocked but there are no additions to index until items have expired. Investigation was undertaken into memory caching with overflow to disk storage [17, 18], but the Geometry Wrapper serialisation was too burdensome for on-demand caching to disk.

Obtaining CRSs, including units of measure, and math transformations can require internet connectivity for lookups from authoritative sources and has been used as justification for only permitting a single CRS in certain contexts [19]. In the context of the Semantic Web, internet connectivity is an underlying principle and typically expected to enable open systems; demonstrated by the SPARQL standard supporting federated querying through HTTP and a typical use case being the publication of data through an HTTP endpoint. Consideration has been made in the implementation for closed systems by allowing users to define the CRSs they require and adding them to the registry.

A practical issue for users of a geospatial system is the conversion of a heterogenous dataset or datasets, with varying CRS or serialisations, to become homogenous. Methods have been included to leverage the interoperability implemented for GeoSPARQL so that users can use GeoSPARQL Jena as an API to undertake these conversions and complement Apache Jena's existing support for serialising triples.

An additional use case is converting existing geospatial data to RDF formats as highlighted for future work in the GeoSPARQL standard. Many platforms and libraries are available for data conversion but are often intended for large scale or diverse formats. A common format for database export or manually prepared data is tabular separated value files, e.g. CSV and TSV etc. Therefore, a tabular file converter has been developed to facilitate the conversion of user geospatial data to RDF. This converter is provided as a pure Java application or API, does not require an external schema or configuration and does not create additional column and row index properties as found in some other conversion libraries. Configuration is through column headings with URIs able to be explicitly stated, formed from a base URI for the file, looked up from common URIs or defined by the user in a separate file.

Datatypes can also be referred in shorthand to standard XSD types [20] or be user defined in file or external prefix file. In RDF the number of properties can vary between individuals of the same class and therefore repeated properties, blank fields, and ragged rows are supported. Apache Jena inferencing and serialisation has been leveraged so that the user can output files for a dataset that incorporates, or separates, inferred triples in a wide range of standards-compliant formats.

4.2. Implementing the Benchmarking Framework

The benchmarking framework was developed in Java and built as a Gradle multi-project. Java was selected due to all three systems providing a Java API and its prevalence in enterprise production systems. A common deployment use case for SPARQL enabled graphstores and databases is as an HTTP endpoint for public querying. All three target systems can be configured as an HTTP endpoint, but Java APIs have been used so that HTTP or network overhead are not factors

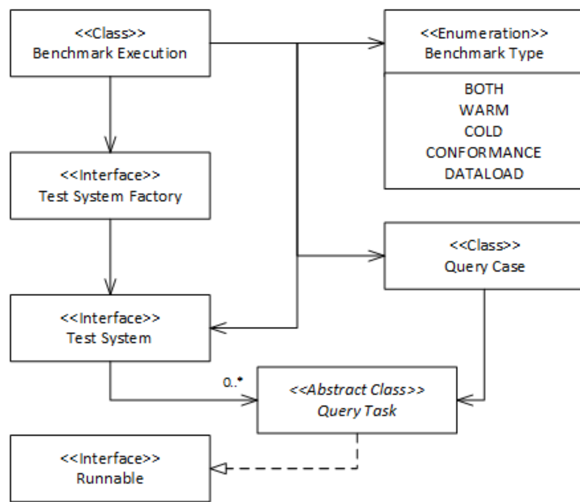

Fig. 2. Class diagram of the main Benchmarking Framework classes and interfaces. The Benchmark Execution requests instances of the Test System from the Test System Factory. The Test System executes Query Tasks for each Query Case that the Benchmark Execution has been set to iterate over and according to the Benchmark Type.

in the results. An area of future work for the framework is the development of querying over HTTP allowing non-Java systems to be compared.

The Gradle build tool was selected to assist in dependency management, distribution, and separation of each system into its own project to avoid dependency conflicts. The Parliament test system uses a legacy version of Apache Jena which conflicts with the more recent version used in the implementation. A core project contains the benchmarking framework for the loading and execution of queries and output of results.

In common with Geographica's approach, a Test System interface is defined which is implemented by the target system to integrate with the framework, see Figure 2. However, in our approach new instances of the Test System are produced using a Test System Factory interface to create separate instances. The initialisation period of these instances is recorded as a separate duration from dataset loading and query execution, again varying from Geographica, and is incurred when a system is initially accessed via the Java API, which may be once, e.g. during start-up for production usage, or multiple times, e.g. during application execution or development.

The execution of queries is performed through a separate Query Task class, varying from Geographica, that is extended by each Test System to standardise the three stages of query execution, i.e. preparation, processing and closure, and recording of durations. This execution standardisation means that all target systems perform the same work and are monitored consistently. The Query Task is executed on a separate thread and is monitored for its duration to limit excessively long query executions.

Investigation was made into incorporating the Java Measurement Harness (JMH) [21] for the benchmarking framework and index size experiments discussed later. JMH is designed to provide consistent and accurate micro-benchmarking by protecting against JVM optimisations and other benchmarking factors. However, the initialization process of Apache Jena conflicted with the benchmarking process of JMH. Therefore, an approach consistent with that adopted by Geographica was followed. This did allow finer reporting of sub-benchmark durations and output results to be developed but an area of future work is the incorporation of JMH.

The results of a query are unpacked in the processing stage as String maps and so decoupled from the SPARQL query engine implementation. The query performance in speed and number of results are summarised to file for comparison across queries and iterations. The executed query is also written to file for cross-comparison and the detailed values returned by the query can be optionally output. The query design in Geographica places filter functions in the SELECT part of the query rather than the WHERE body. This means that even if a function fails due to an exception, a null result is returned for every binding in the WHERE body and can be interpreted as false positives, i.e. a result with no actual data. The framework reports both the number of results returned and the number which contain data, so that comparison can be made and failing queries identified.

The execution of the framework can be performed in four modes: Cold, Warm, Both and Dataload. In Cold mode, each iteration is a new instance of the Test System. The Warm mode, an initialising query is performed on a single Test System which is then reused for the target iterations. The Both mode allows the Cold and Warm modes to be run consecutively. In the Dataload mode, a specified dataset is loading into the Test System's persistent storage for the target iterations with the storage being cleared after each iteration. These modes follow the principles described in the Geographica benchmarking paper [8].

Two default query sets have been defined in the framework, i.e. Geographica Microbenchmark and Geographica Macrobenchmark which are each discussed in more detail later. These query sets are all

loaded from external SPARQL query files and are not programmed into the framework, unlike the design of Geographica. This means that additional query files can be read into the framework for user benchmarking and allows visibility and customisation of the existing queries as standards evolve. The queries relating to a results set are also generated for user inspection.

The Macrobenchmark set utilises additional data which is substituted into queries to provide variation between iterations. This additional data is again provided from external files to allow visibility and customisation. Development of support for using additional data in user-defined benchmarking and conformance testing is an area of future work in the framework. Each usage of the Macrobenchmark produces a set of consistent queries for each iteration across the set but which vary between generation and so between target Test Systems. Reproducibility is achieved by using the output of queries generated for one target system as user-defined queries for testing on subsequent target Test Systems.

The described framework can be extended to provide automated testing of GeoSPARQL standard conformance and other SPARQL standards. This can be achieved by developing a conformance query set and dataset for comparison with the results of the test system. Below is a comparison of the Geographica benchmark query sets with the standard to highlight the areas where conformance testing can be applied:

- Only WKT serialisation is used with no GML serialisation testing.
- Not all Geometry extension non-topological functions are utilised and there is inclusion of Strabon specific functions.
- The RCC8 and Egenhofer relation families are not included and only Simple Feature relations used.
- There is no conversion between different SRS, either geographic or projected, and the datasets all use the default WKT SRS URI.
- There is no testing of the Geometry properties, Topological Vocabulary, Query Rewrite or RDFS Entailment extensions.

Inclusion of these requirements and use cases in a conformance query set allows demonstration of compliance and functionality between test systems and are not covered by the Geographica benchmarking queries. The defining and automated testing of conformance queries is an area of future work for the framework.

5. Implementation Challenges

This section outlines the notable challenges that have been encountered when developing the GeoSPARQLJena library.

5.1. Well Known text (WKT)

The WKT serialisation is a widely used serialisation for geospatial data and referenced in the GeoSPARQL standard. Parsers are available to extract the geometry data from the serialisation for processing. However, the GeoSPARQL standard allows the encoding of Coordinate Reference System URIs as part of the WKT string, which is non-standard and so not compliant with existing parsers. In addition, support of the Geometry Property Extension in the GeoSPARQL standard requires additional metadata for the geometry to be gathered that was not produced by the parsers.

A final issue is the support for more than two coordinate dimensions. The WKT definition permits between 2 and 4 coordinates (XYZM) to be defined but the available parsers only supported two dimensions (XY). The GeoSPARQL standard only considers the X and Y dimensions for spatial operations but it does not explicitly preclude the geometries containing 3 or 4 dimensions from being present in a dataset. Therefore, a WKT parser was implemented to overcome these issues and produce a geometry for the JTS library.

5.2. Geographic Markup Language (GML)

The GML serialisation is an XML based geospatial serialisation that is widely used and referenced in the GeoSPARQL standard. There are several versions with varying cross compatibility and GeoSPARQL implementations are required to state their GML version support. The current implementation of GeoSPARQL-Jena supports the GML Simple Features Profile 2.0, which is a simplified profile of GML 3.2. This profile is intended for compliance with the Simple Features standard and therefore aligns with the GeoSPARQL standard.

GML parsers are available to extract the geometry data but present several issues. The examined parsers are designed to parse GML on a document basis using XML preamble and schema. The GeoSPARQL usage of GML is small isolated fragments of GML geometries that have been encoded within RDF. It was also not possible to locate a GML Simple Feature schema to inform the parser and a comprehensive set of ex-

ample serialisations for the profile were not available. The extraction of geometry metadata and supporting a range of coordinate dimensions, as described in the previous section, was an additional issue affecting the parsing process. Therefore, a specific parser was implemented to overcome the identified issues. While compliance has been sought the development of this parser and support for other GML versions is an area of future work.

5.3. Units of Measure/Buffer and Distance Conversion

It has been outlined previously that each SRS can use one of several unit systems. These can be dependent upon the type of SRS, e.g. *projected* uses linear metres and *geographic* uses non-linear degrees. While a standard international system of units has been established there is still a variety of distance units and sub-units in local usage, such as the standard mile in the United States and United Kingdom. URI definitions have been added for degrees and standard miles, along with their sub-units, for user convenience. Additional units can be registered using the Java Measure API. Conversion between this non-standard and standard units can then be performed consistently.

The GeoSPARQL standard supports *distance* and *buffer* functions which accept arguments in different unit systems, e.g. a geometry in degrees and an expected result in metres. Conversion directly between these units is problematic when considering that a degree in metres for coordinates near the north pole is different to that of coordinates near the equator. The *buffer* function, which expands an enclosing area around a geometry, presents similar difficulties.

These issues are addressed by converting the geometry's SRS to an appropriate SRS for the units, if necessary, upon which the function is applied. For the *buffer* function the resulting geometry is converted back to the original SRS, while the *distance* function ensures the correct units are returned. When the functions are performed on a *projected* SRS but requiring non-linear units a conversion is made to the WGS84 SRS and then the *Euclidean distance* is still found, following the acceptance of error found in the Simple Features [11] and GeoSPARQL [1] standards.

5.4. Relation Families Intersection Patterns

The GeoSPARQL standard describes three Relation Families for identifying spatial relationship between geometries. This includes definitions of the DE-9IM intersection patterns. The Simple Features and GeoSPARQL standards are consistent in stating that the DE-9IM intersection pattern for the *equals* relation, as used by the Simple Features, Egenhofer and RCC8 relation families, is "TFFFTFFFT". However, this does not yield a true result when comparing a pair of point geometries, which are equal. The Simple Features standard states that the boundary of a point is empty. Therefore, the boundary intersection of two points would also be empty.

An alternative intersection pattern of "T*F**FFF*" is used in several spatial libraries [15, 16] and has been applied in the GeoSPARQL-Jena library. This intersection pattern is the combination of the *within* and *contains* relations and yields the expected results for all geometry types.

5.5. getSRID Function

A source of contradiction between the GeoSPARQL and Simple Feature standards is the use of a Spatial Reference System Identifier or SRID. In Simple Features, this provides an integer value for look up against a local catalogue of spatial reference systems and reflecting an internet disconnected design. In GeoSPARQL, the same term is used in the getSRID function, Requirement 5., for the URI string of the geometry literal, which uniquely identifies the spatial reference system and reflects an internet connected design. In seeking cross-compatibility these two different pieces of information, an integer and a URI, are being related together when they are conceptually different. Any future revision to the GeoSPARQL standard would benefit from the two concepts being separated into two functions or renaming the getSRID function.

5.6. GeoSPARQL Schema

The OGC have published the GeoSPARQL v1.0 standard as an RDF/XML schema v1.0.1. This can be imported to provide RDFS and OWL inferencing on a conforming dataset. However, the published schema does not conform with the standard. The property *hasDefaultGeometry* is missing from the schema and instead the *defaultGeometry* property is stated. This prevents RDFS inferencing being performed correctly and has been reported to the OGC Standards Tracker. A corrected version of the schema has been used in the remainder of this work.

5.7. Conformance Dataset & Queries

The GeoSPARQL standard provides several specific query examples with a small dataset. However, these examples are not exhaustive for the general statements of the requirements and in some cases refer to other standard documents, e.g. Simple Features. These standards are highly technical and lengthy making identifying and extracting the required information for an implementation problematic.

Semantic Web technologies are designed to be platform independent. This means that there is opportunity for a compliance dataset and queries to be published to accompany the GeoSPARQL standard. These can then be used or re-purposed by any Semantic Web system on a wide variety of platforms. This would allow GeoSPARQL implementations, and the automated conformance and benchmarking framework discussed previously, to test functionality and evidence compliance more clearly and consistently.

6. Experimental Results and Discussion

This section outlines the evaluation, results, and analysis of the GeoSPARQL Jena implementation against two existing GeoSPARQL systems, Parliament and Strabon. The first set of experiments analyses the effect of varying the size of indexes implemented in GeoSPARQL Jena. This is followed by an overview of the methodology and configuration of the benchmarking framework. The final sections discuss the results obtained for the three systems using the framework.

6.1. GeoSPARQL Jena Index Size

The implementation design described in Section 4 utilises several indexes and registries for the on-demand caching of data. The term *index* has been applied for data that is retained for short-term re-use, e.g. extraction of Geometry Literals for queries, while *registry* is applied to data that has potential long-term re-use, e.g. transformations between Spatial Reference Systems. The registries are expected to be small and persist for the duration of the application while indexes vary in size as data is added and removed through expiry.

The performance benefit and sizing of the indexes are investigated in this section as the size of the indexes presents a compromise between the data extraction process, in-memory storage requirements and retrieval time for existing entries. A test dataset of 100,000 Feature/Geometry and Geometry/LineString triples were randomly generated with GeoSPARQL schema and RDFS inferencing. This dataset was then tested against three queries using the *geof:intersection* and *geo:sfIntersects* functionality and four fixed test values. Each query was designed to test one of the three indexes: Geometry Literal, Geometry Transform, and Query Re-Write.

The queries were performed with a warm up execution followed by ten recorded iterations. Indexes were emptied between each iteration. A range of maximum index sizes were executed from zero, or no indexing, to no maximum size. The expiry time was set to 5 seconds with cleaning occurring at 1 second intervals and was selected based upon experimentation that indicated sufficient duration for data re-use but allowing cleaning to take place during the queries and therefore not retain all generated data.

The results of these experiments are shown in Figure 3. The overall trend in query duration can be seen to fall as the maximum index size is increased. The benefits to each index vary and range between 21.95% improvement for Geometry Literal, 15.13% for Query Re-Write and 5.03% for Geometry Transform. However, the improvement is not continuous as some maximum sizes decline the performance. These are likely caused by retention of values that are generated and not re-used later in the query but preventing more useful values being stored. Therefore, the overall benefit of using the indexes can be seen but the identification of specific maximum sizes is problematic.

It should be noted that identifying the maximum size for the indexes is complicated by background requests using the Geometry Literals during query processing. The test query applied an intersection function that produced 400,000 new Geometry Literal results, although many of these would be empty points, in addition to the 100,000 in the dataset and the 4 test values. During resolving a single query, the Geometry Literal Index received just over 1.7 million requests and peaked in size at 250,348. The peak sizes for the other indexes were Geometry Transform 100,004 and Query Re-Write 200,000, which can be predicted from the dataset and query structure. The queries for these two indexes still required large number of requests to the Geometry Literal Index, which was switched off, and so a balance across all three indexes is required depending upon the functionality utilised.

Any repetition of Geometry Literals in a dataset will also have an impact on the ratio of index max size to

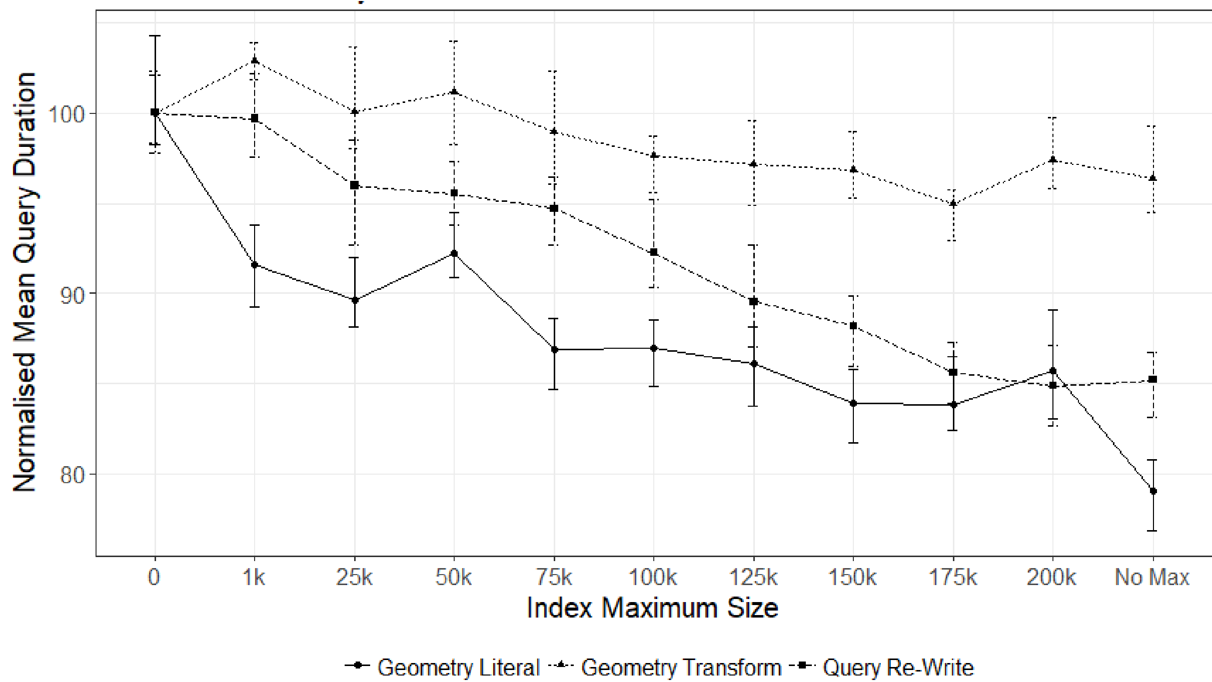

Fig. 3. Mean Query Duration by Index: Mean durations over 10 iterations. Graph shows means normalised by the mean duration of the no/zero size: Geometry Literal: 13.728s, Geometry Transform: 16.839s, Query Re-Write: 41.814s. Error bars show min and max durations.

unique triples. The test dataset had no repeating Geometry Literals, but a dataset can contain non-unique literals yet still consist entirely of unique triples. There would also be variations in query structure and functionality that would affect the number and throughput of items processed and indexed before considering available memory and processing capability.

The benchmarking experiments in the following section were all performed using no max index size and an expiry time of 5 seconds. All benchmarks were executed with limitation to RAM and no memory issues were encountered with this approach. Further work is needed to investigate the effect on performance of dataset size and expiry time in conjunction with the max index size.

6.2. Experimentation configuration and methodology

All benchmarking experiments were performed on the same desktop computer which was a x64 Windows 10 operating system with Intel Core i7-4820K with 16GB and Samsung 850 EVO 500GB. Previously outlined is that all three test systems provide a Java API. Identical Java JVM settings were used for each target system with a maximum heap size of 3GB and parallel garbage collection enabled.

In setting up the different systems there are several configuration points of note. Optimisation of database and graphstore performance can be achieve through tuning various parameters. A general principle has been followed of not optimising to the dataset available with each target system setup *out of the box* following only generally provided guidelines.

In the case of Parliament this meant that tuning parameters were not altered, spatial indexes were created during data loading and RDFS inferencing rules were enabled. GeoSPARQL Jena applied RDFS inferencing during data loading with an automated script applied upon completion. The script is provided by Apache Jena and counts the occurrences of properties and classes in the dataset to provide weightings during query execution.

The Strabon documentation provides several suggested PostgreSQL parameter values for use with different levels of RAM availability. These suggestions were applied but resulted in crashes during execution or exceeded the permitted range of PostgreSQL. Therefore, the default values have been used meaning a different approach to that used in the Geographica benchmark. Inferencing was activated during query execution.

The framework measures the duration of each query execution with three values: query preparation, query processing and their sum for the total duration. For brevity the total durations are reported here, unless specified. A general observation is that the Apache Jena based test systems, Parliament and GeoSPARQLJena, perform more work during iteration over the results in the query processing stage while Strabon spent relatively longer periods in the query preparation stage. This highlights the different strategies deployed by the target systems and the importance of considering the whole execution duration of queries.

In all cases a maximum of one hour was permitted for each iteration. Those cases which timed out are indicated in the text with any non-visible values in graphs being relatively small rather than missing. Each scenario was performed for five iterations.

6.3. Dataset Loading & Initialisation

The six datasets published by Geographica have been used for the Microbenchmark and Macrobenchmark. These datasets use a pre-version 1.0 GeoSPARQL spatial reference system URI which was converted to the version 1.0 URI. This did not change the coordinate values but simply removed a legacy reference from the benchmarking.

The target system each use their own persistent storage method with all supporting multiple graphs. Each dataset was loaded into their own graph within the storage. In the implementation the Apache Jena Java API loading mechanism was used and not the Bulk Loader for large datasets, which may provide faster results. Table 1. shows the data loading and initialisation durations for the target systems.

Table 1
Dataset loading and initialisation durations

	Loading		Initialisation	
Test System	mean (s)	sd	mean (s)	sd
GeoSPARQL Jena	90.694	7.401	0.047	0.003
Parliament	337.976	4.217	0.539	0.041
Strabon	374.439	6.194	80.317	1.882

The loading durations include both the import of triples and any associated one-time spatial indexing setup. The initialisation duration is the duration required to initially access the dataset through the Java API. Therefore, in a production environment this may occur once but in an iterative development environment will be incurred at every execution. In all cases the operations were performed five times with arithmetic mean and standard deviations shown.

The table demonstrates that the implementation, GeoSPARQL Jena, is noticeably quicker at both loading, 1.5 minutes compared to over 5 minutes, and initialising the datasets. This makes it very suited to a development environment when coupled with its minimal setup requirements. Parliament has a more prolonged setting up period when it is building its spatial indexes but is then quick to access prior to querying. Strabon is the slowest to load the datasets and requires a lengthy initialisation period of 80 seconds. This is incurred prior to any query execution and regardless of its data requirements. Therefore, Strabon would be unsuited to a developmental or exploratory environment where an application is being frequently restarted. The following sections discuss the query performance excluding the identified initialization periods.

6.4. Geographica Microbenchmark

The queries executed in this benchmark are those published by the Geographica project and follow the same numbering system. These utilise the non-topological query functions of the Geometry Extension and the filter functions of the Geometry Topology Extension for the Simple Feature relation family and therefore are a subset of potential queries for the GeoSPARQL standard. It was necessary to fix several namespaces in the published queries, but the body of the queries were unchanged. The queries Q6, Q28 and Q29 have been excluded as they contained spatial functions which are not defined in the GeoSPARQL standard and therefore only Strabon could execute.

The results of the microbenchmark are shown for Cold runs (Figure 4), Warm runs (Figure 5) and combined rankings (Table 2). Parliament did not complete any of the queries in the Spatial Joins section within the one-hour time limit. Results are shown in all cases for GeoSPARQL Jena and Strabon but in several cases results were achieved in less than a second. In all cases, except Q15, the number of results returned by all three systems were identical. Q15 uses the distance function to identify geometries within a radius of a fixed point. The difference in results may be attributable to precision of calculations and different approaches to handling the conversion of distance units in query. GeoSPARQL Jena is using the JTS library for calculations which does not reduce precision when other geospatial libraries may do so.

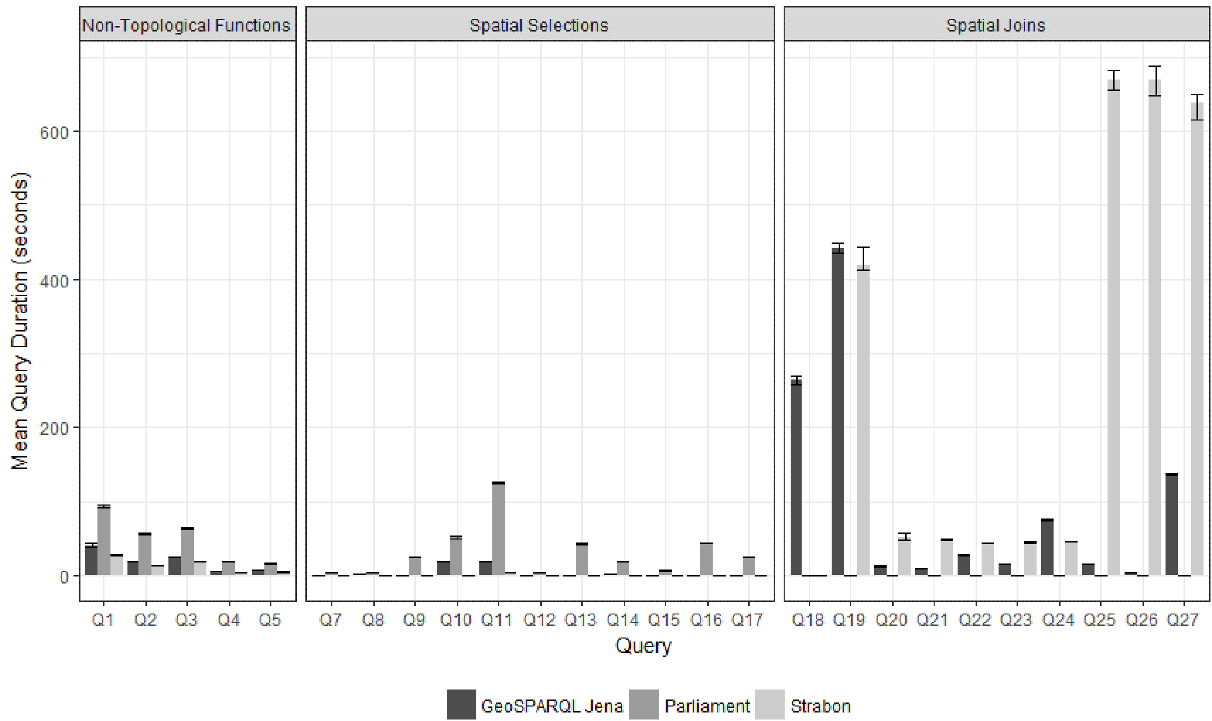

Fig. 4. Mean Query Duration (seconds) by test system microbenchmark in Cold run. Error bars show min/max. Parliament timed out in Q18-Q27.

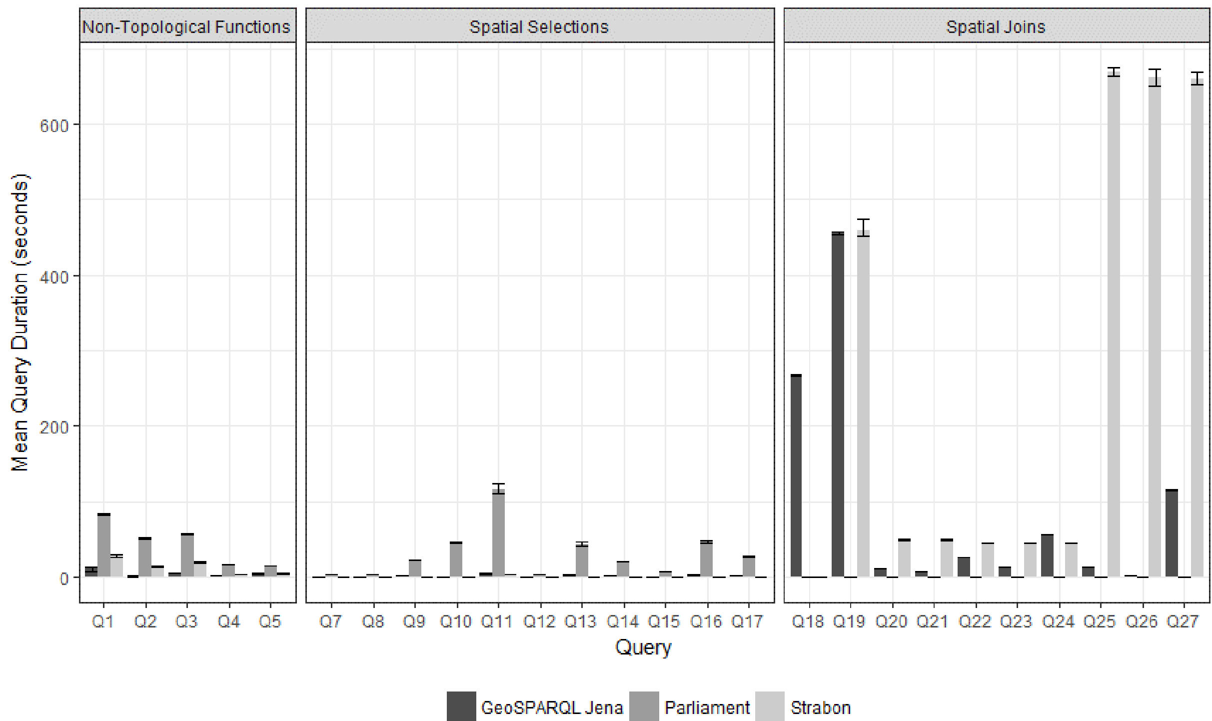

Fig. 5. Mean Query Duration (seconds) by test system microbenchmark in Warm run. Error bars show min/max. Parliament timed out in Q18-Q27.

Table 2
Microbenchmark test system rankings for Cold and Warm runs

	1st		2nd		3rd	
Test System	Cold	Warm	Cold	Warm	Cold	Warm
GeoSPARQL Jena	8	16	18	10	0	0
Parliament	0	0	0	0	26	26
Strabon	18	10	8	16	0	0

The query uses linear *metre* units for distance between two geometries. All geometries in the datasets are in CRS84, which is a geocentric SRS using non-linear *degree* units. In GeoSPARQL Jena, the handling of linear units with non-linear SRS geometries is achieved by temporarily transforming the geometries to a linear SRS. This is necessary as the ratio of *metres* to *degrees* is not fixed and varies according to the latitude of the coordinates. Alternative approaches may use a fixed ratio and tolerate the error as latitude varies.

Another query of note is Q18 which tests for the Simple Features *equals* relation. Strabon is noticeably quicker and completes this in less than 1 second when GeoSPARQL Jena requires just under 4.5 minutes and Parliament times out. This query checks the spatial equality between two graphs numbering 7,551 and 21,993 geometry literals to produce 166 million combinations. Therefore, the speed of processing this scale of combinations by Strabon is noticeable. Further investigation suggests that Strabon is not checking for spatial equality but only lexical equality by matching strings. Spatial equality, but not lexical equality, can exist when two geometries have the same coordinates but use different SRS that reverse the axis order or when one geometry shape has additional coordinates that do not alter the shape, e.g. two straight lines that start and end at the same coordinates, but one has intermediate coordinates. Therefore, the results of Strabon in this query may be accurate but may not fully comply.

A final query of note is Q26 where GeoSPARQL Jena completes in less than 4 seconds while Strabon requires approximately 11 minutes and Parliament times out. This query tests for the Simple Feature *touches* relation, where the boundaries of two geometries align but do not cross into each's interior. The two graphs used are the same graph which contains 325 complex multi-polygon geometries for 105 thousand combinations. GeoSPARQL Jena on-demand indexing will mean that these geometries are readily available while optimisations for this relation, e.g. bounding boxes around the complex shapes, mean that cases can be rejected quickly. Similar optimisations can be applied in Q25 and Q27 and could explain why GeoSPARQL Jena is noticeably quicker than Strabon.

Table 2 shows that Strabon achieves the most frequent fastest completion times in the Cold runs but GeoSPARQL Jena is more often fastest in the Warm runs. This reflects the on-demand indexing implemented in GeoSPARQL Jena. Parliament is consistently the slowest performer and did not complete one set of queries in the time limit. Strabon is generally quicker in the Non-Topological Functions and Spatial Selections queries. These queries utilise a single graph dataset while the Spatial Joins require geometries from two separate graph datasets. This indicates that Strabon is performing optimisations for intra-graph operations, e.g. during the lengthy initialization period, that may not be possible or tractable inter-graph, i.e. the *Curse of Dimensionality*.

Although it would be expected that the Warm runs complete quicker than the Cold runs a comparison between the mean completion times finds this is not always the case with GeoSPARQL Jena 6 cases, Parliament 5 cases and Strabon 10 cases. In many cases the differences are fractional or less than a couple of seconds but the largest are exhibited by Strabon in Q19 and Q27 and GeoSPARQL Jena in Q19. It should also be noted that GeoSPARQL Jena is only marginally slower than Strabon in many queries, expect Q10 and Q11 and Q18 discussed earlier, while there is a noticeably lengthy difference in many queries in the Spatial Joins section for Strabon.

6.5. Geographica Macrobenchmark

The queries executed in this benchmark are those published by the Geographica project and follow the same numbering system. In keeping with the Geographica approach there is variation between query iterations. Each block of queries within a set uses related data but the data selected varies between iterations, e.g. MS0, MS1 and MS2 have consistent data in the first iteration but a different set of data is used in the second iteration. The same resulting queries has been used across each of the test systems to allow direct comparison. This variation between iterations can be seen in Figure 6 where there is more noticeable variation in maximum to mean completion durations. This can partly be attributed to many iterations returning zero results and resolving very quickly.

There is consistency in the number of results returned in all but two queries. Strabon returns zero re-

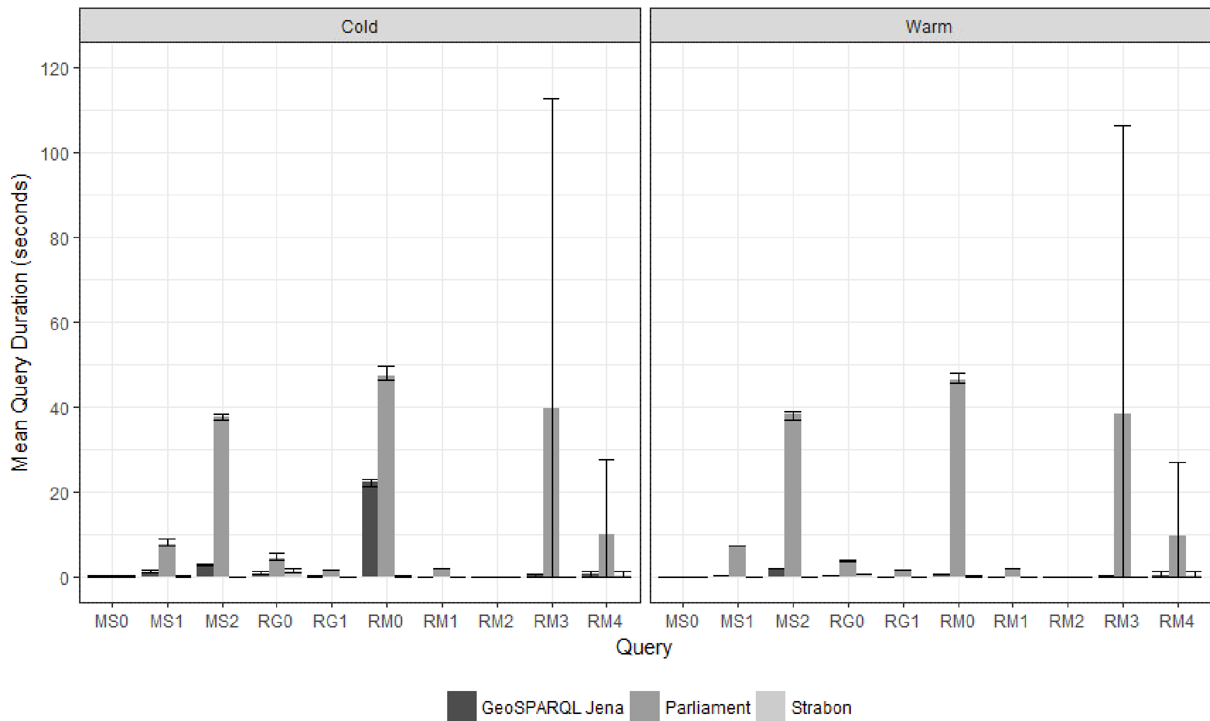

Fig. 6. Mean Query Duration (seconds) by test system for both macrobenchmark runs. Error bars show min/max. Incomplete iterations for Parliament in RM5 and Strabon in MS2, RG1 and RM3. RM5 not shown due to mean completion by GeoSPARQL Jena in 30 mins compared to Strabon in 3.4 seconds (see main text).

sults in four iterations of MS0 when Parliament and GeoSPARQL Jena both return a single result. The second case is MS2 where there is a difference in result count for the only non-zero iteration between Parliament and GeoSPARQL Jena. Strabon did not complete, repeatedly, this iteration and so no completion duration is reported. Strabon also did not complete all the iterations for RM3 and RG1. In all cases where Strabon did not produce results the other test systems had non-zero result counts and Strabon produced an error rather than timing out, suggesting an issue with the processing of results within Strabon.

Parliament timed out after one hour in the RM5 query. This was also the most intensive query for GeoSPARQL Jena and required approximately 30 minutes in each iteration. The RM5 query uses two *intersection* filter functions and a geometry *difference* function with data from two graph datasets. In all iterations GeoSPARQL Jena required a similar duration to complete, even when no results were returned. Strabon was able to resolve the zero results in less than 1 second and only took 15 seconds for the non-zero result iteration. The *intersection* filter function is also used in microbenchmark Q19 and both test systems achieved similar results. This suggests that Strabon's query optimisation selected a resolution strategy which reduced the problem space much quicker than GeoSPARQL Jena.

Further examination of GeoSPARQL Jena finds that the first *intersection*, requiring a cross product of 231.8 million cases, was resolved before the second *intersection* that only required 6,285 cases. If order of resolution is reversed then the former is only 592 thousand cases. SPARQL query optimisation for resolution is controlled by the underlying Apache Jena query engine and persistent storage. Simple manipulation of the query, without changing content, dramatically reduces execution time to approximately 8 seconds, which is quicker than Strabon's result with the original query.

The manipulations required are: 1) moving a *intersection* filter function into the relevant graph clause and 2) reversing the order of the graph clauses. This allows the Apache Jena query optimisation to apply the more efficient resolution strategy. In Strabon, applying the manipulated query returns no results despite it being valid SPARQL 1.1 query. This highlights the different optimisations but also potential non-compliance of the examined query engines.

Table 3
Macrobenchmark test system rankings for Cold and Warm runs

	1st		2nd		3rd	
Test System	Cold	Warm	Cold	Warm	Cold	Warm
GeoSPARQL Jena	6	7	5	4	0	0
Parliament	1	1	3	3	7	7
Strabon	4	3	3	4	4	4

The rankings across the three test systems in Table 3 show that GeoSPARQL Jena achieves the fastest results in the majority of queries. These rankings do not consider the discussed query manipulation for RM5 so Strabon is still ranked as faster than GeoSPARQL Jena in that case. Strabon achieves very quick results in certain cases but did not complete all iterations for three queries. Parliament did not resolve one query and is generally the slowest to complete.

7. Conclusion

This work sets out the design of a fully compliant implementation of the GeoSPARQL standard utilising an RDF graphstore. Previous implementations have achieved partial compliance of a sub-set of extensions. The examined implementations have generally provided geospatial and RDF functionality by extending relational databases requiring additional configuration and setup. Additional design points that have been achieved were Semantic Web standards compliance, minimal configuration and a short initialisation period. This means that the implementation would be suited to both development and production environments.

An on-demand indexing approach was designed and implemented to retain geospatial and supporting data for improved performance. Experimentation was performed to consider the impact of controlling index size and retention periods on query duration. This innovative approach of short and long term caching of key data has been demonstrated to reduce query completion times by up to 20% without incurring initialisation delays. This on-demand indexing also has general utility for other SPARQL applications that require repeated processing of datatypes that are computationally expensive to de-serialise.

To understand the implementation's performance capabilities, a benchmarking framework has also been designed and implemented to perform a comparison with two existing GeoSPARQL systems: Strabon and Parliament. This framework can be expanded for additional test systems through Java interfaces to ensure consistent querying and minimal integration effort.

The design of this framework is based upon importing, processing and reporting on SPARQL queries, rather than a hardcoded queries as developed in the Geographica benchmarking framework for GeoSPARQL. Therefore, it has utility for benchmarking SPARQL queries broader than GeoSPARQL. This approach also provides transparency in the query content and data being benchmarked while variant queries can be easily written and processed. Queries can also be benchmarked on alternative datasets.

An area of future work is extending the benchmarking framework with queries to test GeoSPARQL module conformance and report results. Several areas for conformance testing have been identified and feedback is welcome on further additions. Additional areas include incorporating additional GeoSPARQL systems for testing and allowing user data to be incorporated into SPARQL query templates. Finally, the incorporation of the Java Measurement Harness would provide more robust benchmarking timings. Its usage was investigated but issues with the detail of benchmarking results and conflicts with Apache Jena's initialisation process were not able to be resolved.

The reported benchmarking results show several advantages of the GeoSPARQL Jena implementation over the two benchmarked systems. The dataset loading and initialisation of the implementation are noticeably quicker. In the alternative systems, pre-query spatial index preparation is undertaken that is not incurred by the GeoSPARQL Jena implementation. Despite this, the benchmarking process has demonstrated that the GeoSPARQL Jena implementation has query times comparable or better than the alternative systems in all but one query case (RM5). The GeoSPARQL Jena implementation also completed all the GeoSPARQL benchmarking queries of the Geographica query set.

It has also been identified that minor manipulations to the benchmarking queries can trigger dramatic improvements in query resolution. This was found in the single query case (RM5) where GeoSPARQL Jena was dramatically slower than the Strabon test system. The application of two structural changes to the query, without altering content, reduced query time from 30 minutes to 8 seconds and overtook Strabon's performance.

This demonstrates that SPARQL query writing and optimisation can be very specific to the query engine being utilised. This highlights the challenge in preparing benchmarking queries which do not over accentu-

ate the performance of a specific test system. It further emphasises the need for benchmarking frameworks to be adaptable in processing alternative versions of queries supplied by the user; as designed and implemented in the benchmarking framework.

The GeoSPARQL-Jena [1] implementation, which has been invited for integration as a module by the Apache Jena project, and GeoSPARQL Benchmarking framework [2] have both been published as open source projects.

[1] https://github.com/galbiston/geosparql-jena
[2] https://github.com/galbiston/geosparql-benchmarking

References

[1] M. Perry and J. Herring, GeoSPARQL - A Geographic Query Language for RDF Data, Technical Report, Open Geospatial Consortium (OGC), 2012. http://www.opengeospatial.org/standards/geosparql.

[2] G. Schreiber and Y. Raymond, RDF 1.1 Primer, 2014. https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[3] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, Technical Report, World Wide Web Consortium (W3C), 2013. https://www.w3.org/TR/sparql11-query/.

[4] Y. Guo, Z. Pan and J. Heflin, LUBM: A benchmark for OWL knowledge base systems, *Web Semantics: Science, Services and Agents on the World Wide Web* **3**(2–3) (2005), 158–182.

[5] Oracle Spatial and Graph: Benchmarking a Trillion Edges RDF Graph, Technical Report, Oracle Corporation, 2016. http://download.oracle.com/otndocs/tech/semantic_web/pdf/OracleSpatialGraph_RDFgraph_1_trillion_Benchmark.pdf.

[6] Apache Marmotta, Technical Report, Apache Foundation, 2018. http://marmotta.apache.org/kiwi/geosparql.html.

[7] K. Kyzirakos, M. Karpathiotakis and M. Koubarakis, Strabon: a semantic geospatial DBMS, *The Semantic Web–ISWC 2012* (2012), 295–311.

[8] G. Garbis, K. Kyzirakos and M. Koubarakis, Geographica: A benchmark for geospatial rdf stores (long version), in: *International Semantic Web Conference*, Springer, 2013, pp. 343–359.

[9] M.J. Egenhofer and R.D. Franzosa, Point-set topological spatial relations, *International Journal of Geographical Information System* **5**(2) (1991), 161–174.

[10] A.G. Cohn and J. Renz, Qualitative spatial representation and reasoning, *Foundations of Artificial Intelligence* **3** (2008), 551–596.

[11] O.G. Consortium, Simple Feature Access - Part 1: Common Architecture, 2016. http://www.opengeospatial.org/standards/sfa.

[12] E. Clementini, P.D. Felice and P.V. Oosterom, A small set of formal topological relationships suitable for end-user interaction, in: *International Symposium on Spatial Databases*, Springer, 1993, pp. 277–295.

[13] EPSG Geodetic Parameter Dataset, Technical Report, International Association of Oil and Gas Producers (IOGP), 2018. http://www.epsg.org/.

[14] G.L. Albiston and T. Osman, Semantic Model Assembly Framework for the Generation of Travel Demand, *Manuscript in preparation.* (2018).

[15] Locationtech Java Topology Suite, Technical Report, Eclipse Foundation, 2018. https://projects.eclipse.org/projects/locationtech.jts.

[16] GeoTools - The Open Source Java GIS Toolkit, Vol. 2018. http://geotools.org/.

[17] Apache Commons JCS - Java Caching System, Technical Report, Apache Foundation, 2018. https://commons.apache.org/proper/commons-jcs/.

[18] EH Cache, Technical Report, Software AG USA, 2018. http://www.ehcache.org/.

[19] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen and T. Schaub, The GeoJSON Format, Technical Report, Internet Engineering Task Force, 2016. https://tools.ietf.org/html/rfc7946.

[20] D. Peterson, S. Gao, A. Malhotra, C.M. Sperberg-McQueen and H.S. Thompson, W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, Technical Report, W3C, 2012. https://www.w3.org/TR/xmlschema11-2/.

[21] Java Measurement Harness, Technical Report, OpenJDK, 2018. http://openjdk.java.net/projects/code-tools/jmh/.

[22] R. Battle and D. Kolas, Enabling the geospatial semantic web with parliament and geosparql, *Semantic Web* **3**(4) (2012), 355–370.

[23] M. Zilske, A. Neumann and K. Nagel, OpenStreetMap for traffic simulation (2011).