ExtremeEarth

H2020 - 825258

Deliverable

D3.8

Software for federating big linked geospatial data sources – version 2

Antonis Troumpoukis, Nefeli Prokopaki-Kostopoulou, Giannis Mouchakis and Stasinos Konstantopoulos

June 29, 2021

Status: FINAL
Scheduled Delivery Date: 30 June 2021

Executive Summary

This deliverable is developed in the context of WP3, objective of which is to develop a set of tools for querying, integration and extreme analytics for the big information and knowledge that will be mined from Copernicus data and other auxiliary data sources using the techniques developed in WP2. This information and knowledge will be encoded as linked geospatial data and will be integrated with other open linked data sources to be demonstrated in the two use cases of ExtremeEarth.

Deliverable D3.8 concerns the second phase (months 13-30) of Task 3.4 *federating big linked geospatial data sources*. The aim of the method and software developed in T3.4 is the unified access of linked geospatial data from multiple, possibly heterogeneous, geospatial data servers. For this second phase of development, we have developed a new version of the Semagrow federation engine. The new version has several extensions and adaptations in order to be able to federate multiple geospatial linked data sources, a capability that was, for the most part, missing in the previous version. The results of this task will make Semagrow, currently the state-of-the-art open-source engine for federating geospatial RDF stores, the first engine for federating big geospatial data sources and performing extreme geospatial analytics.

The final experimental evaluation of Semagrow will be reported in the Deliverable D3.5, as it depends on the evaluation framework and datasets currently under development in Task T3.5.

Document Information

Contract Number	H2020 - 825258	**Acronym**	ExtremeEarth
Full title	ExtremeEarth		
Project URL	`http://earthanalytics.eu/`		
EU Project Officer	Riku Leppänen		

Deliverable	**Number**	D3.8	**Name**	Software for federating big linked geospatial data sources – version 2			
Task	**Number**	T3.4	**Name**	Federating big linked geospatial data sources			
Work package	**Number**			WP3			
Date of delivery	**Contract**	30 June 2021		**Actual**	30 June 2021		
Status	Draft ☐ Final ☑						
Nature	Prototype ☑ Report ☐						
Distribution Type	Public ☑ Restricted ☐						
Responsible Partner	NCSR-D						
QA Partner	UoA						
Contact Person	Stasinos Konstantopoulos						
	Email	konstant@iit.demokritos.gr		**Phone**	+30 210 650 3194	**Fax**	n/a

Project Information

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number H2020-825258. The beneficiaries in this project are the following:

Partner	Acronym	Contact
National and Kapodistrian University of Athens Department of Informatics and Telecommunications (Coordinator)	UoA	Prof. Manolis Koubarakis National and Kapodistrian University of Athens Dept. of Informatics and Telecommunications Panepistimiopolis, Ilissia, GR-15784 Athens, Greece Email: (koubarak@di.uoa.gr) Tel: +30 210 7275213, Fax: +30 210 7275214
VISTA Geowissenschaftliche Fernerkundung GmbH	VISTA	Heike Bach Email: (bach@vista-geo.de)
The Arctic University of Norway Deptartment of Physics and Technology	UiT	Torbjørn Eltoft Email: (torbjorn.eltoft@uit.no)
University of Trento Department of Information Engineering and Computer Science	UNITN	Lorenzo Bruzzone Email: (lorenzo.bruzzone@unitn.it)
Royal Institute of Technology	KTH	Seif Haridi Email: (haridi@kth.se)
National Center for Scientific Research - Demokritos	NCSR-D	Vangelis Karkaletsis Email: (vangelis@iit.demokritos.gr)
Deutsches Zentrum für Luft-und Raumfahrt e. V.	DLR	Corneliu Octavian Dumitru Email: (corneliu.dumitru@dlr.de)
Polar View Earth Observation Ltd.	PolarView	David Arthurs Email: (david.arthurs@polarview.org)
METEOROLOGISK INSTITUTT	METNO	Nick Hughes Email: (nick.hughes@met.no)
Logical Clocks AB	LC	Jim Dowling Email: (jim@logicalclocks.com)
United Kingdom Research and Innovation - British Antarctic Survey	UKRI-BAS	Andrew Fleming Email: (ahf@bas.ac.uk)

Contents

List of Figures

List of Tables

1. Introduction

This deliverable is developed in the context of WP3, objective of which is to develop a set of tools for querying, integration and extreme analytics for the big information and knowledge that will be mined from Copernicus data and other auxiliary data sources using the techniques developed in WP2. This information and knowledge will be encoded as linked geospatial data and will be integrated with other open linked data sources to be demonstrated in the two use cases of ExtremeEarth.

Deliverable D3.8 concerns the second phase (months 13-30) of Task 3.4 *federating big linked geospatial data sources*. The aim of this task is the extension and adaptation of federated query processing technology to geospatial data at a very large scale and the integration of these technologies in the Hops data platform. The resulting component will be used to federate geospatial data sources in Strabon, GeoMesa and other geospatial data servers. The work performed for the first phase (months 1-12) of Task 3.4 was reported in Deliverable D3.4. There, we had surveyed the state of the art in federating linked data and geospatial data and analysed the technical challenges that make the literature in ferderated geospatial data so sparse by comparison to federated linked data. We had also prepared a baseline prototype that is not optimized in its execution of federated geospatial joins.

In this deliverable, we present the new version of the *Semagrow federation engine*. The new version has several extensions and adaptations in order to be able to federate multiple geospatial linked data sources, a capability that was, for the most part, missing in the previous version. More specifically, pre-Extreme Earth Semagrow was only able to federate one geospatial store with several thematic stores, and would operate by assuming that all geospatial filters can simply be pushed to the geospatial store. Geospatial data have several characteristics that present several challenges to thematic federation engines, therefore it is important to develop new functionalities that target geospatial data specifically. In order to be able to extend Semagrow's capabilities to federated linked geospatial data, we developed a novel geospatial source selector method, a novel geospatial join optimization technique, and we have extended almost every component of Semagrow's architecture. We have tested these functionalities in two exercises from the Extreme Earth project: (a) linking land usage data with ground observations for the purpose of estimating crop type accuracy; and (b) linking land usage data with water availability data provided for the Food Security use case. Finally, we have integrated Semagrow in Hops as a means for providing a single access point to multiple data sources deployed in the Hops platform.

The results of this task will make Semagrow, currently the state-of-the-art open-source engine for federating geospatial RDF stores, the first engine for federating big geospatial data sources and performing extreme geospatial analytics.

The rest of the deliverable is structured as follows.

- In Chapter 2, we present the state-of-the-art on federated GeoSPARQL query processing at the beginning of the ExtremeEarth project, by summarizing the most important parts from D3.4.
- In Chapter 3, we introduce a a novel source selection method for federated geospatial linked data. This method uses a summary of the spatial extent of each federated data source as a source selection criterion in order to reduce the set of sources that will be tested as potentially holding relevant data.
- In Chapter 4, we introduce a novel join optimization method for federated geospatial linked data. This optimization targets federated geospatial within-distance queries, which are known to be computationally expensive, but can be optimized by adding additional filters with standard topological functions for quickly pruning shapes that are too far away.

- In Chapter 5, we report the development work for the implementation of the new version of Semagrow, which is the result of Task T3.4. Apart from the methodological extensions of Semagrow discussed in the previous chapters, we have extended Semagrow in various ways for processing federated linked geospatial data.
- In Chapter 6, we conclude and we present planned work during the final months of the project. Specifically we present our plans for end-to-end evaluation of the integrated Extreme Earth technologies at the order of magnitude of petabytes.

Finally, in the Appendix we present the queries that were used in the experiments performed in this deliverable.

2. Prior Works

In this chapter, we summarize the state-of-the-art on federated GeoSPARQL query processing at the beginning of the ExtremeEarth project. For a more detailed discussion on the state-of-the-art, please reder to Deliverable D3.4 [21], where we include an extensive survey on federated and geospatial query processing in both the Database domain and the Linked Data domain.

The chapter is structured as follows: in Section 2.1, we present a brief introduction on the GeoSPARQL query language, which is an extension of SPARQL and the de-facto query language for querying Geospatial Linked Data; in Section 2.2, we summarize the state-of-the-art on federated SPARQL query processors; and finally, in Section 2.3 we close with a brief presentation of the Semagrow query federation engine and its pre-ExtremeEarth capabilities.

Throughout the document, we use SPARQL qnames to shorten URIs. The list of URI namespaces that we use are shown in Table 2.1.

2.1 The GeoSPARQL query language

The GeoSPARQL specification [19] defines a set of classes and properties for asserting and querying geospatial information. The `geo:SpatialObject` class comprises any resource that can have a spatial representation. All the usual topological relations (containment, overlap, etc.) are foreseen as properties.

There are two main subclasses of `geo:SpatialObject`: `geo:Feature`, representing geo-located things that exist in the physical world, and `geo:Geometry`, spatial objects that have a single, concrete geographical shape. Each feature is linked with one (or more, representing, e.g., seasonal variation) geometries with the `geo:hasGeometry` property. The `geo:asWKT` property is used to provide the concrete geographical shape of any spatial object as an RDF literal of the `geo:wktLiteral` datatype.[1] Given the above, the link between features and their concrete coordinates follows the pattern:

```
r geo:hasGeometry g . g geo:asWKT "coords"^^geo:wktLiteral .
```

where `r` is an instance of `geo:Feature` and `g` is an instance of `geo:Geometry`.[2]

[1] Two alternative serializations are foreseen by GeoSPARQL, `geo:wktLiteral` and `geo:gmlLiteral`, and two datatype properties, `geo:hasWKT` and `geo:hasGML`. We restrict the discussion in this paper to the WKT serialization, and it is straightforward to transfer this discussion to GML or any other serialization.

[2] Temporal or other restrictions might apply to select the correct `geo:Geometry` instance. We gloss over such considerations that fall well outside the scope of source selection based on geospatial extent.

Table 2.1: List of URI namespaces used throughout the deliverable.

Prefix	Namespace
`rdf:`	`<http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
`rdfs:`	`<http://www.w3.org/2000/01/rdf-schema#>`
`geo:`	`<http://www.opengis.net/ont/geosparql#>`
`geof:`	`<http://www.opengis.net/def/function/geosparql/>`
`uom:`	`<http://www.opengis.net/def/uom/OGC/1.0/>`
`strdf:`	`<http://strdf.di.uoa.gr/ontology#>`
`void:`	`<http://rdfs.org/ns/void#>`
`svd:`	`<http://www.w3.org/2015/03/sevod#>`

Naturally, inference about `geo:wktLiteral` values falls outside RDF graph entailment and can only be performed by specialized geospatial databases. Such entailment is accessed via geospatial functions. For example, the query:

```
SELECT ?r WHERE {
  ?r geo:hasGeometry ?g . ?g geo:asWKT ?w .
  FILTER( geof:sfWithin(?w,"polygon coords"^^geo:wktLiteral) ) .
}
```

uses the `geof:sfWithin` function to access the geospatial operator that computes if the WKT value `?w` retrieved from the graph pattern is contained in another WKT value; such a query fetches all features within a given polygon.

In this context, a geospatial join is essentially a cross product that is being filtered by a geospatial condition comparing values from both expressions. For example, the query:

```
SELECT ?r1 ?r2 WHERE {
  ?r1 geo:hasGeometry ?g1 . ?g1 geo:asWKT ?w1 .
  ?r2 geo:hasGeometry ?g2 . ?g2 geo:asWKT ?w2 .
  FILTER( geof:sfIntersects(?w1, ?w2) ) .
}
```

uses the `geof:sfIntersects` function to access the geospatial operator that computes if the WKT values `?w1` and `?w2` retrieved from the graph pattern intersect; such a query fetches all features that have intersecting geometries.

2.2 Federated SPARQL query processing

Federated query processing consists of three major phases, that is source selection, query planning, and query execution. In this section, we provide a brief presentation of prior work in the literature of federated query processing. For a more extensive discussion on related work, please refer to D3.4 [21].

Recent studies [3, Section 5] find that there is no mature federated GeoSPARQL query processing system. A recent survey that investigates federated query processing and other data integration methods only gives the static transformation of distributed data into a single store as a means of integrating geospatial data [17, Section 4].

2.2.1 Source Selection

The first step in federated SPARQL query processing is to select a subset of the sources that make up a federation for each triple pattern of the query. The goal for the source selector is to prune as many redundant sources as possible in order for the query planner to come up with a more efficient query execution plan.

Most federated SPARQL query processors make use of two basic approaches for their source selection mechanism. In *metadata-assisted source selection*, the federator relies on a dataset descriptor about properties and classes for each federated source (usually expressed using the VoID vocabulary [2]) in order to identify candidate sources for each individual triple pattern of the query. DARQ [22] uses only this method for its source selection. On the other hand, in *metadata-free source selection*, first introduced by FedX [26], the federator identifies the candidate sources by

issuing an ASK SPARQL query to all federated endpoints for each triple pattern of the query. In general, the latter approach is more accurate (because it relies on ASK queries and not in metadata) but the former approach is faster (because issuing an ASK query on the endpoints introduces a significant time overhead). To get the best of both worlds, most federation engines [1, 4, 6, 29] use source metadata to perform a first pruning of the sources and then refine it using ASK queries.

Recent developments in source selection methodology for federated SPARQL systems provided new techniques that consider additional parts of the input query. Thus, *join-aware source selection* focuses not only on individual triple patterns but also on how they join. Sophisticated source metadata about subject and object URI namespaces [24], or about characteristic sets that can describe complete star patterns [18] are able to support inferences on join variables to eliminate redundant sources from the plan.

We will close this subsection with some examples regarding the source selectors discussed above. A VoID-metadata-assisted source selector can decide whether to keep a source d for the pattern `?s p ?o` by simply checking if `p` appears in the metadata of d. On the other hand, for the pattern `?s ?p "str"`, a ASK-based source selector may be more accurate, because the source metadata usually do not include information about every literal that appears in the sources. Finally, consider the join `?s p ?x . ?s p ?y` and a candidate source d for the first pattern. If the subject namespace of all triples of d that match the first pattern is not included in the subject namespaces of all triples that match the second one in all of its candidate sources, then d is a redundant source for the first pattern.

2.2.2 Query Planning and Optimization

The second step in federated SPARQL query processing is to produce a query execution plan by decomposing the original query into several subqueries according to the result of source selection. The goal for the query planner is to produce an efficient plan by arranging the order of subqueries in an optimal way and, if possible, by pushing redundant operators in the source endpoints.

The concept of *exclusive groups* [26] plays an important role in federated query processing. One triple pattern can match either one source or multiple sources. To reduce repetition of sending several triple patterns to a single source, a triple pattern can be grouped with other triple patterns in one subquery. A set of triple patterns that appear in a single source is called an exclusive group, and it has been observed [4, 26] that it is more efficient to group all triple patterns of an exclusive group into a single subquery.

After building subqueries for each source, the query planner arranges the order of subqueries in various combination of execution plans. Even though there are federation engines that calculate the query plan in a greedy fashion [26], most federation engines use statistical information about each of the federated sources for forming the optimal plan [4, 6]. Thus, the cost of a candidate plan is estimated recursively using a *cost model* over statistics about its sub-expressions. Apart from the cardinality of each sub-expression, the cost model in some operations applies a communication overhead, because the intermediate query results during federated query processing are transferred over the network.

Filter pushdown [25] is a standard optimization technique in query processing. The basic idea is that SPARQL filters should be "pushed" as deep in the execution plan as possible, because the query processing time can be reduced by filtering out the data earlier during the evaluation of the query. Especially in the context of federated query processing, by pushing the evaluation of the filters in the source endpoints, we have a reduced query processing time because the results are filtered before to be transferred over the network. Moreover, filter pushdown can be important in the context of GeoSPARQL filters, because each federated GeoSPARQL endpoint has its own spatial index for evaluating geospatial functions, so it makes sense for a federated engine to push these filters in the federated endpoints.

2.2.3 Query Execution

The third step in federated SPARQL query processing is to execute the query execution plan provided by the query planner of the previous step. Thus, the federation engine should implement an evaluation strategy of all SPARQL operators that can appear in the query plan (such as UNION, OPTIMAL, FILTER and so on). Of great importance is the evaluation strategy of federated join operator.

Bind join [7] is an attractive join implementation for distributed and federated environments because it is designed to drastically reduce the communication costs of joining two relations. The idea, similar to the *nested loop join*, is that if we have a very selective outer relation of a join, we can pass the results as bindings to the inner relation in order to filter out a large number of tuples. The difference is that bind join can also work for remote queries since it substitutes the results of the outer relation as bindings to the query of the inner relation.

A naive implementation of a bind join [6, 22] can be highly inefficient since it will create and execute a different query for each result of the outer relation. A more elaborate solution for reducing the overall number of queries produced is to group multiple bindings into a single query [4, 26]. Fortunately, SPARQL 1.1 specification foresees a VALUES keyword as a mechanism for passing multiple variable bindings at once. In addition, Schwarte et al. [26] proposed a technique that can simulate this behavior using a more complex UNION expression in order to support legacy SPARQL 1.0 endpoints. An important component for the implementation of a bind join evaluation strategy is a SPARQL *query executor*. Given a SPARQL endpoint, a SPARQL query and a set of bindings, the query executor prepares the SPARQL query to be issued to the endpoint (either using the VALUES expression or the UNION transformation), makes the connection with the endpoint, issues the query, and fetches the result set to the federator.

2.3 The Semagrow query federation engine

In this section, we discuss the state of Semagrow at the beginning of the Extreme Earth project. In particular, we summarize the relevant techniques used in each step of the federated query processing.

Source Selection Pre-ExtremeEarth version of Semagrow uses a hybrid source selection for individual triple patterns of a SPARQL query. In particular, it exploits source metadata encoded in the Sevod vocabulary [11] to perform a first pruning of the set of sources for each triple pattern, and then refines this set by issuing ASK queries to the source endpoints.

Query Planning Semagrow uses a dynamic-programming-based query planner and exploits statistical information about each of the federated sources encoded in the Sevod vocabulary [11] and a cost model for forming the optimal plan of a federated query. Moreover, it uses both exclusive group and filter pushdown optimizations for reducing the communication cost. Finally, the optimizer is re-engineered with a query graph model (similar to that proposed by Hartig and Heese [9]) in order to support more complex queries than simple joins of triple patterns and FILTER expressions.

Query Execution Semagrow uses an efficient implementation of bind join similar to that of FedX [26] for supporting SPARQL 1.0 endpoints. Moreover, the query execution module of Semagrow uses a re-engineered architecture to allow executor plugins that manage syntactic and semantic heterogeneity. In particular, apart from the standard query executor for federating SPARQL

data sources, it allows federating data sources that may offer other APIs, such as CassandraQL API. Thus, even though that Semagrow processes SPARQL queries, it can appropriately re-write the sub-queries for each data source by filling in the missing expressivity (e.g. arbitrary joins for CassandraQL sources) [10].

At the beginning of Extreme Earth, neither Semagrow nor any other federated query processor provided a full, mature support for federating geospatial data sources [3]. As a result, a federation (as per the state of the art at the beginning of Extreme Earth) could include any number of thematic stores but only a single geospatial store.

3. Geospatial Source Selection

In this chapter, we present the first methodological extension of Semagrow developed in Task T3.4, by introducing a novel source selection method for federated geospatial linked data. In particular, we explore the idea of annotating data sources with a bounding polygon that summarizes the spatial extent of the resources in each data source, and of using such a summary as an (additional) source selection criterion in order to reduce the set of sources that will be tested as potentially holding relevant data. To this end, we present our source selection method and also present and discuss experiments that evaluate two different types of summaries (bounding boxes and bounding polygons) against not using geospatial summaries. The evaluation draws on data and queries from the use cases of the Extreme Earth project and shows our method to substantially improve query processing time.

3.1 Introduction

Geospatial linked data brings into the scope of the Semantic Web and its technologies a wealth of datasets that combine semantically-rich descriptions of resources with these resources' geo-location. There are, however, various Semantic Web technologies where technical work is needed in order to achieve the full integration of geospatial data, and federated query processing is one of these technologies.

Source selection is the first step for most *federated query processors*, mapping triple patterns in the query to a subset of the SPARQL endpoints that make up a federation. Source selection is typically based on characteristic properties and URI namespaces to eliminate sources that do not have relevant data and dramatically improve the efficiency of federated query processing. This approach breaks down when geospatial datasets are distributed by geographical extent.

In this chapter, we explore the *spatial extent* of each data source as a new type of summary. Spatial extent makes more sense for geospatial data, by comparison to the vocabularies and URI namespaces used which makes more sense for thematic data. In practice, we investigate how to best exploit that geospatial datasets are likely to be naturally divided in a canonical geographical grid (consider, for example, weather and climate data) or following administrative regions or, more generally, areas of responsibility (consider census data as an example).

In the remainder of this chapter, we first present a characteristic use case which we use throughout the chapter (Section 3.2). In Section 3.3, we describe our geospatial source selector that uses endpoint metadata (in the form of a bounding shape that contains all shapes that appear in the endpoint) to filter out sources that do not contribute to the result. We then use our open-source implementation of this source selector to empirically compare the efficiency of using bounding-box descriptions, precise shape descriptions, and conventional source selection (Section 3.4). Finally, present and discuss relevant work (Section 3.5) and conclude (Section 3.6).

3.2 Motivation and Use Case

We will now present a characteristic use case that both motivates some of our technical choices and backs our experimental setup with data and a query load: the *food security* use case of the Extreme Earth project. In this use case, crop type information needs to be combined with nearby snowfall and snow storage, since irrigation largely depends on snow storage and seasonal release of fresh water. [1]

[1] Cf. `http://earthanalytics.eu/food-security-use-case.html`

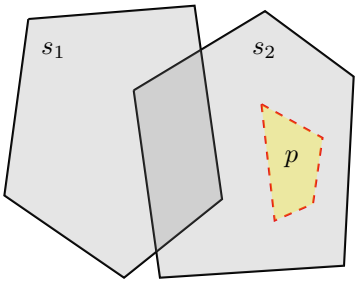

Figure 3.1: The boundaries of two endpoints s_1 and s_2, with a polygon of interest p that lies within the boundary of s_2.

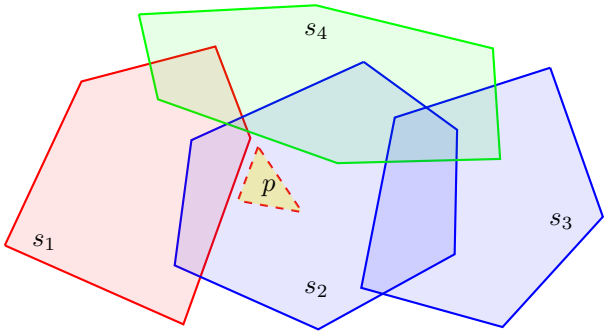

Figure 3.2: Four endpoints, where s_1 uses the 'red' vocabulary to describe resources, s_2 and s_3 the 'blue' vocabulary, and s_4 the 'green' vocabulary.

The queries that are most relevant for this analysis are spatial *within* queries, spatial *intersection* queries, and *within-distance* queries: retrieving the land parcels with a given crop that are within, intersecting, or within a given maximum distance from any snow-covered area, without requiring the exact distance. Notice that within-distance queries are considerably more computationally demanding than spatial overlap and inclusion queries that can be answered from the index. On the other hand, by comparison to queries that actually compute the distance, they offer themselves to aggressive optimization: Many instances can be discarded in advance as too far away to be within the required distance so that the (expensive) distance computations actually performed by the database are minimized.

Consider, for example, Figure 3.1. Evaluating a filter that only retains shapes within distance d from p can immediately (i.e., from the database index) discard all shapes contained in s_1 if the distance between s_1 and p is greater than d. This presents a huge optimization opportunity by comparison to computing the distance between p and all shapes in s_1 and then comparing these against d. Transferring this discussion to federated query processing, we see that geospatial datasets are often published by public administrations or other entities with responsibility over a specific geographic extent. This motivates applying this optimization to the source selection level: if s_1 and s_2 were the bounding polygons of all resources served by two GeoSPARQL endpoints, then source selection can exclude s_1 from the execution plan.

Naturally, a GeoSPARQL query will normally combine geospatial restrictions with thematic triple patterns; in our case, for example, referring to a crops codelist or hierarchy. Consider, now, Figure 3.2 where each of the four data sources only contains triples using a specific codelist or vocabulary. Such a situation is likely to appear when different organizations publish data regarding different aspects of a geographical region (for example, crops and precipitation data), some of which are also independently published for each region. For a query using the green and blue vocabularies to retrieve entities of interest within a given distance from p, it makes no sense to consider including s_1 in the execution plan. This partitioning is amenable to conventional source selection based on metadata about the vocabularies used in each data source. However, in order to have the optimal source selection a federation engine would need to also exclude s_3 based on its geospatial extent, and only query s_2 and s_4. Such a source selection can only be achieved by extending conventional federated source selection with a mechanism that combines metadata about the thematic content of a data source with metadata about its geospatial extent.

3.3 Method

In this section, we describe the proposed geospatial source selection method in detail. We begin with some preliminaries, we then describe the required source metadata, we continue with the presentation of the algorithm, and we close with a discussion on the implementation.

```
_:d rdf:type void:Dataset ;
    void:sparqlEndpoint <http://localhost:30300/Query> ;
    svd:boundingWKT
      "<http://www.opengis.net/def/crs/EPSG/0/4326> POLYGON ((9.5 46.4,
      9.5 49.0, 17.1 49.0, 17.1 46.4, 9.5 46.4))"^^geo:wktLiteral .
```

Figure 3.3: Example of metadata for the geospatial source selector.

3.3.1 Preliminaries

In this subsection, we present some preliminary definitions. These definitions will help us to properly define our source selector in the following subsections.

Let a and b be two spatial objects, that can be points, lines and/or polygonal areas. We say that a and b are *disjoint* iff they do not have any points in common. Moreover, we say that a *contains* b iff no points of b lie in the exterior of a, and at least one point of the interior of b lies in the interior of a. Notice that in this case, a does not contain its border, but a does contain itself. We say that a and b *touch* iff they have at least one boundary point in common, but they share no interior points.

Let Q be a GeoSPARQL query and F be the set of sources federated by a query processor. We say that $S \subset F$ is the *optimal source set* for Q if (a) using only the sources in S gives the same results for Q as using all the sources in F; and (b) there is no subset of S that gives the same results for Q as using all the sources in F.

3.3.2 Source metadata

The proposed geospatial source selector requires from every federated source to be tagged with the following information:

Definition 1. Let s be a geospatial source. Then, a *bounding polygon* $\mathcal{B}(s)$ of s is any polygon that *contains* every spatial object that appears in s.

For representing the bounding polygon of a source s, we define the following property:

Definition 2. The bounding geometry of all geometries in a dataset can be denoted using the predicate `svd:boundingWKT` as follows:

```
svd:boundingWKT  rdf:type     rdf:Property   ;
                 rdfs:domain  void:Dataset   ;
                 rdfs:range   geo:wktLiteral .
```

An example of such an annotation is shown in Figure 3.3. In this example, a source accessible from a specific endpoint contains only geometries within a specific polygon, denoted as a WKT literal.

3.3.3 Source selection algorithm

In this subsection, we describe our source selection algorithm in detail. We assume the existence of some helper routines that their implementations is not included as a separate algorithm. Given a GeoSPARQL query Q, the routine GeospatialFilters(Q) returns the set of all GeoSPARQL

filters that appear in Q. Given a filter f, the routine VARS(f) returns all variables that appear in f. Given a filter condition without free variables c, EVAL(c) returns true if the condition holds, otherwise false.

Our source selector is designed to support a family of GeoSPARQL queries. We concentrate on simple filter expressions that consist of a geospatial function call. In particular, we consider filters that their condition is of the form $r(x, y)$ or of the form `geof:distance`(x, y, u) op d, where r is a non-disjoint geospatial function (i.e., one of the functions `geof:sfEquals`, `geof:sfIntersects`, `geof:sfTouches`, `geof:sfContains`, `geof:sfOverlaps`, `geof:sfWithin`, and `geof:sfCrosses`), x, y are variables or WKT literals, u is unit of measure, op $\in \{<, \leq, =\}$, and d is a numeric literal. Our method can be easily extended for the other GeoSPARQL functions apart from the simple features relation family (e.g., Egenhofer and RCC8).

Algorithm 1 takes as input a GeoSPARQL filter f and a set of bindings $B = \{v_1/b_1, \ldots, v_n/b_n\}$ for each free variable $v_1, \ldots, v_n$ of f, and returns true if the condition of f does not hold if we substitute v_i in f with any possible shape contained in b_i, for all $1 \leq i \leq n$, otherwise it returns false. The idea is that if two shapes x, y are disjoint, then every shape contained in will be also disjoint from every shape x contained in y. Moreover, if x touches y, then there does not exist any shape contained in x that is equal, is within, or contains any shape contained in y and vice-versa. Finally, if the distance between x and y is greater than D, then the distance between every shape contained in x and every shape contained in y is also greater than D. We call this algorithm BPFILTEREMPTY, because it can check whether a GeoSPARQL filter f returns an empty result set, provided that all bindings of every variable of the filter are contained within a known polygon.

Algorithm 2 prunes the set of sources for every triple pattern of the form $(x, \texttt{geo:asWKT}, o)$, by using Algorithm 1, the relevant GeoSPARQL filters of the triple pattern and the bounding polygons of each candidate source. Let f be a geospatial filter of a GeoSPARQL query Q. We distinguish two cases for f, based on the number of its free variables. First, suppose that f contains a single variable o. Then, Q should contain a triple pattern $t = (x, \texttt{geo:asWKT}, o)$. Let s be a candidate source for t. If we consider s in the evaluation of Q, then all bindings for o that come from s are contained within the bounding polygon $\mathcal{B}(s)$ of s. Assume that BPFILTEREMPTY$(f, \{o/\mathcal{B}(s)\}) =$ true. It is easy to see that the result set of Q in this case will not contain any bindings for o that come from s; therefore, s is not contained in the optimal source set of t and can be pruned. Second, suppose that f contains two variables o and o'. Then, Q should contain two triple patterns $t = (x, \texttt{geo:asWKT}, o)$ and $t' = (x', \texttt{geo:asWKT}, o')$. Let s be a candidate source for t. As previously, we want to use Algorithm 1 to check whether s can be pruned. However, in this case, we want to take into consideration the bindings of o' as well, which come from the set of the candidate sources of t'. Notice that all bindings for o' are contained within the *union* of the bounding polygons for all candidate sources of t'. Therefore, s can be pruned if BPFILTEREMPTY$(f, \{o/\mathcal{B}(s), o'/\mathcal{B}(s')\}) =$ true for every candidate source s' of t'. We repeat this process several times for all filters of Q until the set of sources cannot be further pruned.

Algorithm 3 makes use of the previous algorithms and defines the proposed source selection mechanism. Notice that ASWKTSOURCESELECTOR targets only triple patterns that link a specific geometry URI with its WKT serialization, i.e., patterns of predicate `geo:asWKT`. It leaves out triple patterns that link a resource with its geometry (i.e., `geo:hasGeometry` predicates), and triple patterns of thematic information. As discussed in Section 3.2, the geospatial pruning obtained by Algorithm 2 should be complemented by a thematic source selector. Here we use the (extended) thematic source selector of Semagrow (cf. Subsection 3.3.4). As a final note, the result of the algorithm is a mapping that associates each the triple pattern of the query to a subset of the set of the sources of the federation. Since, in practice, source selectors are build 'on top' of each other, our source selector takes such a mapping as input, which can be the output of another source selector.

Algorithm 1 BPFilterEmpty

Input: a GeoSPARQL filter f s.t. $\textsc{Vars}(f) = \{v_1, \ldots, v_n\}$, and a set of bindings $B = \{v_1/b_1, \ldots, v_n/b_n\}$.
Output: true or false

f' is a GeoSPARQL filter obtained by f by substituting $v_1, \ldots, v_n$ with $b_1, \ldots, b_n$.
if $f' =$ `FILTER(geof:`$r(x,y)$`)`, where $r \in \{$`sfEquals, sfWithin, sfContains`$\}$ **then**
  return $\textsc{Eval}($`geof:sfDisjoint`$(x,y)) \vee \textsc{Eval}($`geof:sfTouches`$(x,y))$
else if $f' =$ `FILTER(geof:`$r(x,y)$`)`, where $r \in \{$`sfOverlaps, sfCrosses, sfTouches, sfIntersects`$\}$ **then**
  return $\textsc{Eval}($`geof:sfDisjoint`$(x,y))$
else if $f' =$ `FILTER(geof:distance`(x,y,u) `op` $d)$, where u is unit of measure, `op` $\in \{<, \leq, =\}$, and d is a numeric literal **then**
  return $\textsc{Eval}($`geof:distance`$(x,y,u) > d)$
else
  return false
end if

Algorithm 2 AsWktSourceSelector

Input: a GeoSPARQL query Q, a set T of triple patterns, a set S of sources, a mapping $\sigma : T \to 2^S$, a mapping $\mathcal{B} : S \to B$ s.t. $\mathcal{B}(s)$ is the bounding polygon of s
Output: a mapping $\sigma : T \to 2^S$

$F \coloneqq \textsc{GeospatialFilters}(Q)$
repeat
  $\sigma_{\text{old}} \coloneqq \sigma$
  if there exists some $f \in F$, $t \in T$, and $s \in S$ such that $\textsc{Vars}(f) = \{o\}$, $t = (x,$ `geo:asWKT`$, o)$, $s \in \sigma(t)$ and $\textsc{BPFilterEmpty}(f, \{o/\mathcal{B}(s)\})$ **then**
    $\sigma(t) \coloneqq \sigma(t) - \{s\}$
  end if
  if there exists some $f \in F$, $t, t' \in T$, and $s \in S$ such that $\textsc{Vars}(f) = \{o, o'\}$, $t = (x,$ `geo:asWKT`$, o)$, $t' = (x',$ `geo:asWKT`$, o')$, $s \in \sigma(t)$ and for all $s' \in \sigma(t')$ it holds $\textsc{BPFilterEmpty}(f, \{o/\mathcal{B}(s), o'/\mathcal{B}(s')\})$ **then**
    $\sigma(t) \coloneqq \sigma(t) - \{s\}$
  end if
until $\sigma_{\text{old}} = \sigma$
return σ

Algorithm 3 GeospatialSourceSelector

Input: a GeoSPARQL query Q, a set T of triple patterns, a set S of sources, a mapping $\sigma : T \to 2^S$, a mapping $\mathcal{B} : S \to B$ s.t. $\mathcal{B}(s)$ is the bounding polygon of s, thematic metadata $\mathcal{P}$
Output: a mapping $\sigma : T \to 2^S$

$\sigma \coloneqq \textsc{AsWktSourceSelector}(Q, T, S, \sigma, \mathcal{B})$
$\sigma \coloneqq \textsc{ThematicSourceSelector}(T, S, \sigma, \mathcal{P})$
return σ

Figure 3.4: Geospatial source selector algorithm.

3.3.4 Implementation

We provide an implementation of our geospatial source selector using the rdf4j framework[2] with its additional geospatial functionality. For our experiments we use Semagrow to plan and execute the federated queries. Our implementation extends the thematic Semagrow source selection mechanism which uses (a) the pre-ExtremeEarth source selector of Semagrow which uses predicate metadata and ASK queries to perform source selectors of individual triple patterns; and (b) a new source selector implemented during ExtremeEarth, similar to that of HiBISCuS [24], which uses URI-prefix metadata for join-aware source selection. More information on the development carried out on Semagrow during ExtremeEarth can be found in Chapter 5 (cf. Section 5.1 for the source selection part).

3.4 Evaluation

In this section, we evaluate the performance of the proposed source selector. We describe in detail the experimental setup and we analyze the results. Our experiment is based on real-world datasets and is inspired by a practical use-case scenario in the domain of food security.

3.4.1 Experimental Setup

Datasets For the experimental evaluation, we use the following data sources:

1. The Database of Global Administrative Areas (GADM) for Austria[3], which contains all administrative divisions of Austria up to Level-3.
2. The Austrian Land Parcel Identification System (INVEKOS)[4], which contains the geo-locations of all crop parcels in Austria and the owners' self-declaration about the crops grown in each parcel.
3. A snow cover map which contains thematic and geospatial snow data within the Danube catchment from February to April of 2018 provided from our partner VISTA [5]

The datasets are publicly available as shapefiles, and are transformed into RDF with the GeoTriples tool [15]. We partition the above data in 27 smaller datasets according to the polygons of the states of Austria.[6] For each of the 9 states of Austria, we create 3 datasets; a dataset of the administrative regions of the state, a dataset of the crop parcels that are contained in the state, and a dataset of the snow cover areas that are contained in the state. If a crop parcel or a snow area belongs to more than one state, we split it into several parts so that each part fits entirely in a single state and we place each part to its corresponding dataset. Finally, we modify the URIs of all resources so that all resources that appear in the same dataset share a common prefix, which is unique among the prefixes of all datasets. Table 3.1 illustrates the statistics for the datasets of the experiment; in the table, we group the datasets by type and we display statistics about the sum of the group, as-well-as average and standard deviation for each dataset in the group. Notice that we have a large standard deviation for each group; this is due to the fact that the states of Austria are unequal in size. We use the Strabon geospatial RDF store [13] for serving the data.

[2] Cf. `https://rdf4j.org/`
[3] Cf. `https://gadm.org/maps/AUT.html`
[4] Cf. `http://www.data.gv.at/katalog/dataset/f7691988-e57c-4ee9-bbd0-e361d3811641`
[5] Cf. `http://earthanalytics.eu/food-security-use-case.html`
[6] The code that we used for partitioning the data is publicly available in `https://github.com/semagrow/semagrow-geotools`. The RDF datasets used in the experiments can be found in `http://rdf.iit.demokritos.gr/dumps/gss`.

Table 3.1: Dataset statistics.

datasets	#datasets		#triples total	#triples geospatial	#triples thematic	#properties
gadm1-9	9	total	57,087	2,231	54,856	35
		average	6,343	248	6,095	35
		stdev	4,888	185	4,703	0
crops1-9	9	total	14,056,959	2,008,137	12,048,822	7
		average	1,561,884	223,126	1,338,758	7
		stdev	1,254,534	179,219	1,075,315	0
snow1-9	9	total	331,190	66,238	264,952	5
		average	36,799	7,360	29,439	5
		stdev	27,349	5,470	21,879	0

Federations For the experimental evaluation, we use two federations; one with 27 source endpoints (i.e., one endpoint per dataset); and one with 19 source endpoints (i.e., one endpoint for each administrative and crop dataset, all snow datasets in a single endpoint). For each federation, we set up 3 Semagrow federators, which is configured with a different source source selector. The federators of them-27 and them-19 use the standard thematic source selection of Semagrow, while the remaining federators use a geospatial source selection on top of it. The difference between geo-mbb-27, geo-mbb-19, geo-poly-27, and geo-poly-19 is that in the first two, each source is tagged with the minimum bounding box of all shapes that appears in the source, while in the last two with the actual polygon of the corresponding areas. This increased accuracy, however, leads to an increased metadata size. (cf. Table 3.2) Regarding the evaluation of GeoSPARQL queries, Semagrow uses an exclusive group optimization (for triple patterns that belong to the same source); geospatial joins are evaluated using a bind join evaluation strategy with a filter push-down optimization.

Queries In Table 3.3, we give the queries of the experiment, the number of triple patterns of the query (#tp), the number of geospatial selection filters, i.e., geospatial filters with one free variable (#gs), the number of geospatial join filters, i.e., geospatial filters with two free variables (#gj), and the size of the result set of the query (#r). Queries Q1 and Q2 differ on the polygon they are parameterized with. Notice that the queries use a small number of sources of the federation. For every query, we define its *administrative part* as the triple patterns that refer to administrative data (i.e., datasets gadm1-9), its *crop part* as the triple patterns that refer to crop data (i.e., datasets crops1-9), and its *snow part* as the triple patterns that refer to snow data (i.e., datasets snow1-9). Q1 and Q2 comprise only an administrative part, Q3 (resp. Q4) comprises an administrative and a snow (resp. crop) part, Q5 and Q7 comprise a snow and a crop part, Q6 and Q8 contains all three types of parts. Q1, Q2, Q5-8 use the `geof:sfIntersects` function; Q3, Q7, and Q8 use the `geof:distance` function; and finally, Q4, Q6-8 use the `geof:sfWithin` function. The queries of the experiment can be found in Section A.1 of the Appendix.

Experiment deployment and execution We use a Kubernetes 1.14 cluster with 1 master node and 8 worker nodes with a total if 120 cores and 264GB RAM. All queries are executed

Table 3.2: Federations used in the experiment.

	them-27	geo-mbb-27	geo-poly-27	them-19	geo-mbb-19	geo-poly-19
Source selection	thematic	geospatial	geospatial	thematic	geospatial	geospatial
metadata #triples	2988	3015	3015	2719	2738	2738
metadata file size	260 KB	269 KB	1.9 MB	108 KB	113 KB	1.6 MB
#datasets	27	27	27	19	19	19

Table 3.3: Queries used in the experiment.

	Query	#tp	#gs	#gj	#r
Q1	Municipalities intersecting a given WKT	6	1	0	5
Q2	Municipalities intersecting another given WKT	6	1	0	5
Q3	Snow cover within 5 km from a given municipality.	9	0	1	22
Q4	Potato fields within a given municipality.	9	0	1	203
Q5	Snow-covered potato fields intersecting a given WKT	10	2	1	208
Q6	Snow-covered potato fields within a given municipality.	14	0	3	38
Q7	Potato fields within 5km from snow cover and intersecting a given WKT	10	2	1	1464
Q8	Potato fields within 5km from snow cover and within a given municipality.	14	0	3	12

three times; the execution times reported are the average of the second and third run. Experiment deployment and execution is done through the KOBE benchmarking engine [12], and the KOBE configurations for reproducing the experiments are publicly available.[7]

3.4.2 Experimental Results

Table 3.4 contains the results of the experiment. All average times are in seconds. The relative standard deviation is at most 0.15 for each measurement. We display the following evaluation metrics: to check if the source selection time overheads are recovered by reduced planning and execution time, we display the average source selection (ssel), planning (plan), and execution (exec) times for the second and the third run of the query; to evaluate the efficiency of the pruning of each source selection, we display the number of sources that are accessed during the evaluation (#s); to check if any source selector achieves optimal pruning, we include the size of the optimal source set (#opt) for each query. Finally, to verify that no source selector removes necessary sources, we display the number of returned results (#r), and we (cf. Table 3.3 for the expected result set size). Notice that in some queries the execution evokes an error (err), and in these situations the federator returns no results. In the remaining cases though, all federators return the correct number of results.

Comparison of query processing times

Regarding source selection time, we observe that them-27 and them-19 (in short them) are the fastest ones; then we have geo-mbb-27 and geo-mbb-19 (in short geo-mbb); and finally we have geo-poly-27 and geo-poly-19 (in short geo-poly). This happens because the source selector of geo-mbb and geo-poly wraps the source selector of them. Moreover, the sources in geo-poly are annotated with polygons, which are more complex shapes than the bounding boxes in geo-mbb. Thus, the boundary comparisons performed by the geospatial source selection are slower in geo-poly. This difference is more pronounced in Q3, Q7, and Q8 which use the `geof:distance` function, which is computationally costlier than the containment and intersection functions used in the rest of the queries.

Regarding query planning and query execution time, the geo-poly federators are faster than geo-mbb federators, which are faster than them. Only in geo-poly we obtain a complete evaluation of all queries; 3 (resp. 5) queries fail to be processed in geo-mbb (resp. them) due to errors in the execution phase. These errors occur when a federator issues a huge workload of source queries to the

[7] The experiment specifications can be found in `https://github.com/semagrow/kobe` as a part of the geofedbench suite of KOBE.

Table 3.4: Experimental results.

		geo-poly-27					geo-mbb-27					them-27				
	#opt	time (s)			#s	#r	time (s)			#s	#r	time (s)			#s	#r
		ssel	plan	exec			ssel	plan	exec			ssel	plan	exec		
Q1	1	0.2	0.05	0.05	1	5	0.2	0.05	0.05	1	5	0.1	0.05	0.5	9	5
Q2	1	0.2	0.05	0.05	1	5	0.2	0.05	0.1	2	5	0.1	0.05	0.5	9	5
Q3	3	1.6	0.1	6.7	4	22	0.2	0.1	7.4	5	22	0.1	0.1	12.6	10	22
Q4	2	0.3	0.1	0.1	2	203	0.2	0.1	err	6	-	0.1	0.1	err	10	-
Q5	2	0.7	0.2	0.2	2	208	0.2	0.3	0.2	2	208	0.1	0.3	err	18	-
Q6	3	1.5	14.4	0.2	3	38	0.3	15.1	err	11	-	0.2	16.6	err	19	-
Q7	2	8.2	0.3	11.4	2	1464	0.2	0.5	11.7	2	1464	0.2	0.5	err	18	-
Q8	4	8.0	17.1	1.4	5	12	0.3	17.3	err	7	-	0.3	17.3	err	19	-
		geo-poly-19					geo-mbb-19					them-19				
	#opt	time (s)			#s	#r	time (s)			#s	#r	time (s)			#s	#r
		ssel	plan	exec			ssel	plan	exec			ssel	plan	exec		
Q1	1	0.2	0.05	0.05	1	5	0.2	0.05	0.05	1	5	0.1	0.05	0.4	9	5
Q2	1	0.2	0.05	0.05	1	5	0.2	0.05	0.1	2	5	0.1	0.05	0.4	9	5
Q3	2	0.8	0.1	1.5	2	22	0.1	0.1	1.5	2	22	0.1	0.1	1.5	2	22
Q4	2	0.3	0.1	0.2	2	203	0.2	0.1	err	6	-	0.1	0.1	err	10	-
Q5	2	0.5	0.2	0.2	2	208	0.2	0.3	0.2	2	208	0.1	0.3	err	10	-
Q6	3	0.9	14.5	0.3	3	38	0.3	15.6	err	7	-	0.1	14.1	err	11	-
Q7	2	6.9	0.2	11.7	2	1464	0.2	0.3	11.5	2	1464	0.1	0.2	err	10	-
Q8	3	6.5	14.2	2.0	3	12	0.3	15.4	err	5	-	0.2	14.4	err	11	-

endpoints, and as a result, the sources are not able to serve all these requests. Moreover, we observe that there exists a correlation between the number of sources and the effectiveness of planning and execution. Having a large number of sources per triple pattern requires the construction of a larger query plan, which clearly affects the time for producing it; this is highlighted in Q6 and Q8 which have 14 triple patterns. An overestimation of the number of sources results to a large number of source queries because the query executor poses queries to irrelevant sources, and because the optimizer cannot group all triple patterns that appear in a single source into a single source query. The number of source queries affect not only the completion of the execution, but the execution time as well; e.g., notice that Q2 in geo-poly the execution is two times faster than that in geo-mbb because in geo-mbb more source queries are required in order to evaluate the result.

Finally, notice that in general, the query processing time is faster in them-19, geo-mbb-19, and geo-poly-19. This happens because the federation has a smaller number of endpoints (i.e., 19 instead of 27).

Comparison of source selection pruning

Regarding them, the source selector exploits the thematic information (i.e., properties and URI-prefixes) of the sources and assigns to the administrative (resp. crop, snow) part of the query only the administrative (resp. crop, snow) sources. Moreover, in Q3, Q4, Q6 and Q8, the administrative part of the query is further restricted to a single administrative source, because the pattern that specifies the name of the municipality appears in a single administrative source. In the federations with 19 datasets we have only one snow source, thus the snow part of the query is assigned in a single source. This explains the optimal pruning of them-19 in Q3.

The selectors of geo-poly and geo-mbb achieve better performance by exploiting the geospatial knowledge of the sources. Q1, Q2, Q5, and Q7 have geospatial selection filters, parameterized with a polygon within a single state of Austria. This explains the optimal pruning for geo-poly. In Q2,

geo-mbb outputs 2 sources instead of 1, because unlike Q1, Q5 and Q7, the polygon of Q2 is also contained in the bounding box of a neighbor state.

In Q3, Q4, Q6 and Q8, similarly to them, geo-poly and geo-mbb restrict the administrative part of the query in the source of the state of interest. Then, the geospatial selectors try to prune all sources that refer to irrelevant states according to the geospatial filters of the query. Regarding Q4, Q6, and Q8, geo-mbb prunes all crop sources that their bounding box is disjoint from the bounding box of the state of interest, while geo-poly, being more accurate, does better by keeping only the crop sources that refer to the state of interest, because the source boundaries do not overlap. Regarding Q3 and Q8, geo-poly-27 keeps only the neighboring snow sources of the state of interest; it does not achieve optimal pruning, though. The given municipality appears in the east border of its belonging state, but the source selector cannot exclude the possibility of its position being on the other border (or even towards the center), because the exact geometric shape of the municipality is discovered only during query execution. Even in this case though, geo-poly-27 is more accurate than geo-mbb-27, because the distance between the bounding box of the state of interest and the bounding box between a non-neighboring state happens to be less than 5km.

Discussion

In this experiment scenario, the most effective approach is to use a geospatial source selection parameterized with the exact boundaries of each source. By doing so, we achieve optimal pruning for geospatial filters that contain only standard spatial relations (Q1, Q2, Q4-7) and we achieve the best pruning for distance filters (Q3, Q8), but we spend slightly more time in source selection. We may get a decrease in source selection time and metadata size if we use minimum bounding boxes instead. But in this case, the planing and execution phase is slower and, even worse, the federated endpoints could face a huge query load, making the query processing impossible (Q4, Q6, Q8).

3.5 Related Work

The literature on federated geospatial query processing is very sparse, in either the Semantic Web community, the geographical information systems community, or the wider databases community. Recent studies [3, Section 5] find that there is no mature federated GeoSPARQL query processing system. A recent survey that investigates federated query processing and other data integration methods only gives the static transformation of distributed data into a single store as a means of integrating geospatial data [17, Section 4].

The only related system is *SkyQuery* [16], a federation of astronomy databases. SkyQuery optimizes the execution of SQL-like queries based on metadata similar in nature to the VoID dataset statistics, provided by the individual databases when registering to the federation. *Cross match queries*, in particular, use containment constraints to retrieve objects corresponding to the same astronomical body allowing for some error in the measurements. The constraint is satisfied by looking up an index of spherical triangles, operating analogously to our bounding WKTs: for each triangle the index points to all databases that hold objects within the triangle.

In fact, the triangles that make up the SkyQuery index are organized as a containment hierarchy, analogously to how R-trees are used in geospatial database indexes. This higher level of detail (by comparison to our system) allows SkyQuery to use its index not only for source selection but to fully optimize query execution.

3.6 Conclusion

We presented a source selection method that combines the thematic source selection typically used in federated query processing with an additional data source filtering based on the bounding WKT that summarizes the geospatial extent of all resources in a data source. The implementation of our method is provided as open source, integrated with the Semagrow federated GeoSPARQL processor.

We explored two alternative bounding WKTs: the minimum bounding box and the minimum bounding polygon. The former is straightforward to compute and achieves more efficient source selection run-times; the latter is more precise in excluding unneeded sources, so that the sources in the federation are not burdened by pointless querying. Experimental results show that our method has substantial positive impact on the federated sources' burden; and that for distance queries, using the bounding polygon has a considerably higher source selection time (which is not always recovered at planning and execution time) than using the bounding box. On the other hand, using the bounding polygon always terminates; while using the bounding box may inundate the sources with queries to the point of failure. The overall conclusion is that precise bounding polygons should be preferred since: (a) the source selection run-time is (partially or fully) recovered by shorter planning and execution run-time; and (b) more precise selection makes the federation engine more prudent with the Web resources it consumes.

4. Geospatial Join Optimization

In this chapter, we present the second methodological extension of Semagrow developed in Task T3.4, by introducing a novel join optimization method for federated geospatial linked data. This optimization targets federated geospatial *within-distance* queries, i.e., queries where we want the distance between two shapes that are in different sources to be less than a specific distance. The distance operator is computationally expensive, since it cannot be answered from the spatial index. Thus, in our optimization we try to rewrite each subquery by inserting additional geospatial filters with standard spatial relations, in order to exploit the spatial index of the GeoSPARQL endpoints and help them with the evaluation of the query. In this chapter, we introduce and evaluate our optimization technique. The experimental setup draws on data and queries from the use cases of the Extreme Earth project and shows our method to substantially improve query processing time.

4.1 Introduction

Geospatial linked data brings into the scope of the Semantic Web and its technologies a wealth of datasets that combine semantically-rich descriptions of resources with these resources' geo-location. There are, however, various Semantic Web technologies where technical work is needed in order to achieve the full integration of geospatial data, and federated query processing is one of these technologies.

Query optimization is a feature of many database management systems. The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans. In the context of federated query processing, the most important factor that reduces the query processing time is the number of intermediate results from the subqueries issued by the federator to the remote endpoints. Thus, most federated systems employ techniques to reduce the number of intermediate results, such as efficient join ordering and filter pushdown optimizations. However, especially in the context of geospatial data, we may have situations where a we have queries that return the same number of results, but have major difference in query processing time. For instance, consider a geospatial dataset that contains all points of interest of Greece, and the queries: "retrieve all POIs that their distance from Syntagma Square is less than 1 km" and "retrieve all POIs that are within Athens and their distance from Syntagma Square is less than 1 km" [1]. The within operator is much faster than the distance operator, because it uses the spatial index. Thus, the second query is much faster, because it includes a fast pruning of all POIs outside Athens, and reduces the search space for the within distance operator.

In this chapter, we present an optimization for *within-distance* federated geospatial joins. For our optimization, we assume a bind join-like physical implementation with a filter pushdown. That is, if a query contains a two-variable geospatial function where WKTs from multiple endpoints are needed, the federator fetches "left-hand" WKTs to partially bind two-variable functions and then it pushes the filter to "right-hand" endpoint. Notice that the filter pushed in "right-hand" endpoint contains a single free variable, because the other variable is bound with WKTs that have been already retrieved. Therefore, in the case of within-distance filters (which cannot be answered by the spatial index), the federation can speed-up the evaluation by adding another geospatial filter as a first step (which can use the spatial index) in order to prune all WKTs that are too far away to be within the required distance, so that the "right-hand" query to be evaluated faster by the federated endpoint.

In the remainder of this chapter, we first present a characteristic use case which we use throughout the chapter (Section 4.2). In Section 4.3, we describe our geospatial join optimization for federated within-distance queries in detail. We implement this technique in Semagrow and then we compare

[1] The Syntagma Square is located in the center of Athens.

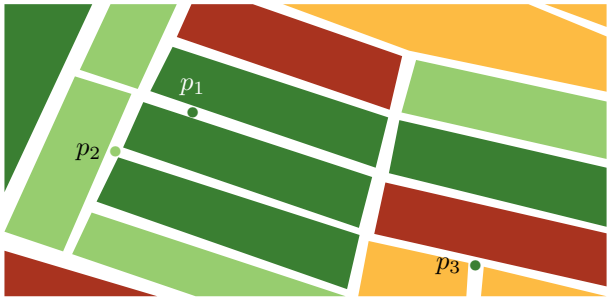

Figure 4.1: 3 ground observations located in the roads adjacent to field parcels, used for crop-type valitation. p_1, p_2 provide a positive and p_3 a negative validation.

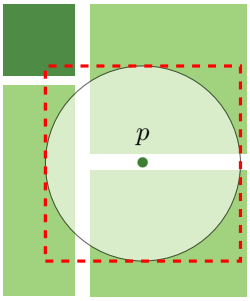

Figure 4.2: A buffer of size d around p and its minimum bounding box.

the optimized version with the unoptimized one (Section 4.4). Finally, we present and discuss relevant work (Section 4.5) and conclude (Section 4.6).

4.2 Motivation and Use Case

We will now present a characteristic use case that both motivates some of our technical choices and backs our experimental setup with data and a query load: the *validation* of food security data. In this use case, crop type information needs to be validated with nearby ground observations [20].

Detailed land usage data is crucial in many applications, ranging from formulating agricultural policy and monitoring its execution, to conducting research on climate change resilience and future food security. Land usage can be inferred from Earth Observation images or collected through self-declaration, but in either case needs to be validated against land surveys. The standard approach for this validation is to match each instance in the land survey dataset (GPS points) with the nearest land parcel (a GIS shape) and compare the crops observed in the survey against the crops declared or inferred for the matching parcel.

Although conceptually straightforward, in operational scenarios this rule can be misleading. Ground observations are geo-referenced to a point on the road adjacent to the field, which is often ambiguous in agricultural areas with several adjacent parcels; further exacerbated by GPS accuracy. However, a more sophisticated (and also computationally demanding) approach can estimate the error rate of the land usage data: for every survey point there must be at least one parcel with the same label in reasonable proximity; otherwise at least one nearby parcel is mis-labelled (although we cannot automatically infer which one).

The task of the crop-data data validation is performed as follows: A ground observation is irrelevant to the analysis if it is more than 10 meters from any crop parcel. For the remaining ground observations, if the closest crop parcel from the ground observation has the same crop type with the ground observation, then the ground observation provides a positive validation; if the crop types do not match, the ground observation provides a negative validation. In Figure 4.1 we illustrate a collection of crop parcels and 3 ground observations, where p_1 and p_2 provide a positive and p_3 provide a negative validation. Such a computation requires retrieving the crop parcels within a given maximum distance from a given ground observation, ordering them by ascending distance and finally taking the closest crop parcel.

In a federated scenario, where the ground observations are stored in a different source endpoint than the crop parcels, the analysis requires a federated within-distance operator for retrieving the candidate parcels. As discussed in Section 4.1, the computation of the federated within-distance part of the query can be sped up by discarding in advance all crop parcels that are too far away from the given ground observation. In Figure 4.2 we illustrate a point p and its nearby crop

parcels. Notice that all candidate parcels what are within distance D intersect with a buffer of size D around p. Notice also, that every parcel that does not intersect with the bounding box of this buffer is not a candidate parcel. Since the bounding box of the buffer is less complex shape than the buffer (which is a circular shape), we can use the bounding box for pruning all shapes that are too far away from the given ground observation p.

4.3 Method

In this section, we describe the proposed geospatial join optimization method in detail. We begin with an overview of our approach, we continue with some preliminaries and the presentation of the algorithm, and finally we close with a discussion on the implementation.

4.3.1 A simple example

Our optimization technique focuses on within-distance federated geospatial joins. Such queries contain filters of the form:

```
FILTER( geof:distance(?x, ?y, uom) < d ) .
```

where `uom` is a unit of measure URI, `d` is a numeric literal, `?x` belongs to the one part of the join, and `?y` belongs to the other part. Moreover, we assume that the federation engine evaluates all federated joins in a bind join fashion with a filter pushdown optimization. Therefore, it is not hard to observe that in such a case, the query that fetches the right part of the geospatial join should contain the above filter, and that one of the two arguments of `geof:distance` (e.g., the variable `?y`) should be bound by the variable bindings of each result of the query that fetches the left part of the join.

In a naive implementation of bind join, for each result of the left part of the join, the federator engine would issue to the source endpoint of the right part of the join a query that contains a filter of the following form:

```
FILTER( geof:distance(?x, WKT_LITERAL, uom) < d ) .
```

Such queries are usually slow, because the evaluation of the distance operator cannot benefit from the spatial index of the source. However, we can help the evaluation of the source query in the remote endpoint by adding an additional geospatial comparison for filtering out all irrelevant geometries:

```
FILTER( geof:sfIntersects(?x, BBOX_BUF_LITERAL) ) .
```

where `BBOX_BUF_LITERAL` is the minimum bounding box of a buffer of size d around `WKT_LITERAL`. The bounding box filter is evaluated by the geospatial source using the spatial index, thus giving fast access to a subset of shapes, in which the exact distance filter is then applied to. Notice that this optimization can be performed during query execution time (and not before evaluating the query), because the `WKT_LITERAL` is not known at planning time.

This approach can be extended in order to apply in more elaborate implementations of bind join, where multiple bindings can be passed in a single query. In Subsection 4.3.3, we will define our technique in a more general way to handle the extended case.

4.3.2 Preliminaries

In this subsection, we present some preliminary definitions. These definitions will help us to properly define our join optimization in the following subsection.

Let a and b be two spatial objects, that can be points, lines and/or polygonal areas. We say that a and b *intersect* iff they share any portion of space. Moreover, we say that a *contains* b iff no points of b lie in the exterior of a, and at least one point of the interior of b lies in the interior of a. Let d be a length measurement. Then, a *buffer* of size d around a is a spatial object that contains all points that their distance from a is less or equal to d. Finally, the *minimum bounding box* of a is a rectangle whose sides are parallel to the x and y axises and minimally enclose a; it is formed by the minimum and maximum (x, y) coordinates of a.

Let us now focus on GeoSPARQL queries. A *variable binding* is a pair u/v, where u is a variable and v is an RDF value. Each element of the result set of a GeoSPARQL query is a *query binding*, which is a set of variable bindings. Let b be a query binding. We say that u is a *bound* variable w.r.t. b if b contains a variable binding u/v. In this case, we write $b(u)$ to denote the RDF value v. Moreover, we say that u is a *free* variable w.r.t. b iff u is not bound w.r.t. b. Finally, let S be a set of query bindings and u be a variable. We say that u is a free (resp. bound) variable w.r.t. S if is a free (resp. bound) w.r.t. to all query bindings $b \in S$. For the rest of the paper, when we will refer to a binding we will mean a query binding.

4.3.3 Join optimization algorithm

In this subsection, we describe our join optimization algorithm in detail. We assume the existence of some helper routines that their implementations is not included as a separate algorithm. Given a geometric literal a, a numeric literal d and a unit of measure u, the routine Buffer(a, d, u) returns a geometric literal that represents a buffer of size d around a measured in units of type u. Given a geometric literal a, the routine Mbb(a) returns a geometric literal that represents the minimum bounding box of a. Given a SPARQL endpoint E, a GeoSPARQL query Q and a set of bindings B, the routine EvaluateRemoteQuery(E, Q, B) implements the behavior of the query executor (cf. Subsection 2.2.3) and returns a set of bindings obtained from the source. Recall that the set of bindings B is actually a subset from the bindings of left part of the bind join.

Our goal is to extend EvaluateRemoteQuery with the optimization that we have sketched in Subsection 4.3.1, adapted for the bind join implementations that use one query for multiple bindings. Thus, in Algorithm 4, we define EvaluateRemoteQueryOpt, which behaves as a wrapper for the original behavior of the query executor.

The first step is to identify whether the input query is in the form of our interest. We consider queries that have geospatial filters that their condition is of the form `geof:distance(?w1,?w2,u)` $< d$, where u is a unit of measure URI and d is a numeric literal. Moreover, we require that only one of the arguments `?w1`, `?w2` to be bound from the bindings of left part of the join; on the remaining cases our technique is not applicable. If both arguments of the filter are free, the geospatial join is to be evaluated in the source; if both arguments are bound, the evaluation of the filter requires from the source to calculate the distance between known geometric literals. When the optimization is not applicable, EvaluateRemoteQueryOpt falls back to the original EvaluateRemoteQuery behavior.

The next step is to rewrite the query. After we identify that the free variable is the variable x and the bound variable is the variable y, we add an additional filter that forces x to intersect with the minimum bounding box of the buffer of size d around y. This filter should prune all shapes that can be easily recognized to have a distance greater than d from y. Since we have several bindings for the variable y, we cannot hard-code the bounding box directly in the query (as we

Algorithm 4 EvaluateRemoteQueryOpt

Input: a GeoSPARQL endpoint E, a GeoSPARQL query Q, and a set of bindings B.
Output: a set of bindings R

if $Q =$ `SELECT` E_1 `WHERE {`
E_2
`FILTER( geof:distance(?w1,?w2,`u`) <` d `) .`
E_3
`},`
where E_1, E_2, E_3 are SPARQL expressions,
u is a unit of measure URI, d is a numeric literal,
one of `?w1`, `?w2` is a free variable w.r.t S (denoted as x), and
one of `?w1`, `?w2` is a bound variable w.r.t S (denoted as y) **then**
  $z \coloneqq$ a fresh variable
  $Q' \coloneqq$ `SELECT` E_1 `WHERE {`
E_2
`FILTER( geof:sfIntersects(`x`,`z`) )`
`FILTER( geof:distance(?w1,?w2,`u`) <` d `)`
E_3
`},`
  $B' \coloneqq \{b \cup \{z/\text{Mbb}(\text{Buffer}(b(y), d, u))\} : b \in B\}$
  $R' \coloneqq \text{EvaluateRemoteQuery}(E, Q', B')$
  $R \coloneqq \{r - \{z/v\} : r \in R'\}$
  return R
else
  return $\text{EvaluateRemoteQuery}(E, Q, B)$
end if

Figure 4.3: Geospatial join optimization algorithm.

have done in Subsection 4.3.1), but we use an additional fresh variable z. Therefore, the condition of the additional filter now becomes `geof:sfIntersects(`x`,`z`)`, and we need to provide additional bindings for the new variable z. This is done by extending each binding b from B with the variable binding z/v, where v is obtained by calculating the minimum bounding box of the buffer of size d around the value that b assigns in y.

The final step is to use the original EvaluateRemoteQuery with the rewritten query and the extended bindings to obtain the result set of the query. Since the original query did not contain the newly constructed variable z, we need to remove through a post-processing all variable bindings that refer to z from the result set.

4.3.4 Implementation

We provide an implementation of our optimization technique using the rdf4j framework[2] with its additional geospatial functionality. For our experiments we use Semagrow to plan and execute the federated queries. Our implementation extends the existing Semagrow query execution mechanism with a GeoSPARQL query executor that uses the optimization discussed previously. More information on the development carried out on Semagrow during ExtremeEarth can be found in Chapter 5 (cf. Section 5.3 for the federated join optimization part).

[2] Cf. `https://rdf4j.org/`

Table 4.1: Dataset statistics.

dataset	#triples	#entities	#shapes	#properties
INVEKOS	14,056,959	4,010,514	2,008,137	7
LUCAS	30,379	8,650	4,325	11

4.4 Evaluation

In this section, we evaluate the performance of the proposed optimization. We describe in detail the experimental setup and we analyze the results. Our experiment is based on real-world datasets and is inspired by a practical use-case scenario in the domain of food security.

4.4.1 Experimental Setup

Datasets For the experimental evaluation, we use the following data sources:

1. The Austrian Land Parcel Identification System (INVEKOS)[3], which contains the geo-locations of all crop parcels in Austria and the owners' self-declaration about the crops grown in each parcel.
2. The EUROSTAT's Land Use and Cover Area frame Survey (LUCAS)[4], which contains agro-environmental and soil data by field observation of geographically referenced points.

The datasets are publicly available as shapefiles, and are transformed into RDF with the GeoTriples tool [15]. The LUCAS dataset was extended with a set of mappings [20] between LUCAS land cover codes and INVEKOS crop types. Table 4.1 gives more details about these datasets. We illustrate the number of shapes that appear the original shapefiles, as-well-as the number of triples, entities, and properties that appear in the transformed RDF datasets. Each dataset is loaded on a different SPARQL endpoint; we use the Strabon geospatial RDF store [14] for serving the data.

Federations For the experimental evaluation, we use two federations of the above datasets; one Semagrow federation in which the execution engine uses technique discussed in Section 4.3 (semagrow-opt) and one that does not use it (semagrow-std). Regarding the evaluation of federated geospatial joins, each Semagrow federation uses a bind join evaluation strategy with a filter push-down optimization.

Queries In Table 4.2 we summarize the queries of the experiment. Each row of the table corresponds with a query template; the parameter is shown in the "parameter" column of the table. Q1-3 are derived from the data validation use case (cf. Section 4.2), and are used to estimate the reliability of the INVEKOS dataset. For each LUCAS instance URI, we check if: (Q1) the closest INVEKOS instance is under 10 meters away and their crop labels match (positive validation); (Q2) the closest INVEKOS instance is under 10 meters away and their crop labels do not match (negative validation); or (Q3) there is no INVEKOS instance within 10 meters (neutral validation). Instead of simply providing a boolean result, we formulated the queries as SELECT queries, so that the data analyst can get more information regarding the corresponding instances. Thus, each

[3] Cf. http://www.data.gv.at/katalog/dataset/f7691988-e57c-4ee9-bbd0-e361d3811641
[4] Cf. https://esdac.jrc.ec.europa.eu/projects/lucas

Table 4.2: Queries used in the experiment.

	parameter	query	#tp	characteristics
Q1	LUCAS URI	return the nearest INVEKOS instance if it is within 10 meters and their crop types match	10	Subquery, ORDER, LIMIT 1
Q2	LUCAS URI	return the nearest INVEKOS instance if it is within 10 meters and their crop types do not match	10	Subquery, ORDER, LIMIT 1, FILTER NOT EXISTS
Q3	LUCAS URI	return the LUCAS instance if there is no INVEKOS instance within 10 meters	10	Subquery, ORDER, LIMIT 1, FILTER NOT EXISTS
Q4	distance	return all INVEKOS instances within D meters from a LUCAS instance	5	-

query either returns a single result (if the LUCAS instance has the desired property) or an empty result set (otherwise). Notice that these queries have several characteristics apart from federated within-distance geospatial joins (cf. Section 5.2 for a discussion on the complexity of the query set). In order to focus on the behavior of the within-distance geospatial join, we have also included Q4. This query returns all INVEKOS instances that are within a given distance from a specific LUCAS instance. The query is parameterized with various distances ranging from 10 meters to 100 kilometers. As for the LUCAS instance, we used the one that is closest to the center of the minimum bounding box of Austria. The queries of the experiment can be found in Section A.2 of the Appendix.

Experiment deployment and execution We use a Kubernetes 1.14 cluster with 1 master node and 8 worker nodes with a total if 120 cores and 264GB RAM. All queries are executed three times; the execution times reported are the average of the second and third run. Experiment deployment and execution is done through the KOBE benchmarking engine [12], and the KOBE configurations for reproducing the experiments are publicly available.[5]

4.4.2 Experimental Results

In the following, we will discuss the experimental results of our evaluation. We first discuss the improvement of the execution time of the query set obtained from the data validation task (Q1-3) and then we analyze the performance of our optimization technique for various distances (Q4).

Data Validation Query Set (Q1-3)

In the first part of the experimental study, we compare the optimized (semagrow-opt) with the unoptimized (semagrow-std) version of Semagrow using the query load obtained by the data validation task. Table 4.3 contains the experimental results. For every query template, we illustrate: the number of queries for each template (#queries), the number of the queries that return result (#results), and the query processing time for semagrow-std and semagrow-opt. For each federation,

[5] The experiment specifications can be found in `https://github.com/semagrow/kobe` as a part of the geofedbench suite of KOBE.

Table 4.3: Experimental results for Q1-3.

			query processing time			
			semagrow-std		semagrow-opt	
	#queries	#results	total	average	total	average
Q1	2488	1650	83 hours	120 sec	106 mins	2.6 sec
Q2	2488	398	82 hours	119 sec	99 mins	2.4 sec
Q3	2488	400	81 hours	117 sec	74 mins	1.8 sec

we display the total time to evaluate the query load and the average time for each query of the query load.

Regarding the data validation task (although irrelevant for the evaluation of the optimization technique), we observe that LUCAS provides 66% positive validation points, 15% negative validation points, and 16% irrelevant points for INVEKOS, based on the returned results of the queries. What is important though from a practical point of view is the time required for the validation task. Thus, we notice that the queries are much faster if we use the optimized version of Semagrow. The unoptimized version would require several days for the evaluation, while with our optimization technique the task reduces to several hours.

Even though the query set contains several complex characteristics, the bottleneck for the query evaluation is the calculation of the within-distance operator, since the execution plan as follows: For Q1 and Q2, Semagrow retrieves the WKT of the given point from LUCAS (which is a fast operation since it requires fetching information from a single instance), and then it retrieves all parcels that are within 10 meters from this WKT (together with their distance and crop type) from INVEKOS. Then, Semagrow sorts the INVEKOS parcels according to their distance and takes the first one. Since the distance is only 10 meters, the parcels are very small in number, therefore the sorting operation is very fast as well. Then, Semagrow retrieves the point crop type from LUCAS and compares it with the crop type of the parcels. Again, this operation is fast because it requires retrieving information from a single instance. Similarly, for Q3, Semagrow retrieves the crop type and WKT of the given point from LUCAS (a fast retrieval of information from a single instance), and checks if there is any point within 10 meters from this WKT from INVEKOS.

Within-distance operation from a given LUCAS point (Q4)

In the second part of the experimental study, we compare the optimized (semagrow-opt) with the unoptimized (semagrow-std) version of Semagrow using a query of fetching all INVEKOS parcels that are found within increasing distance from a specific LUCAS point. Table 4.4 contains the experimental results. For every instance of the query template Q4, we illustrate: the distance parameter of the within-distance operation, the query processing time and the number of results for semagrow-std and semagrow-opt. Moreover, we display the number and percentage of shapes that are pruned from INVEKOS by the additional filter that the optimization process places in the source query that corresponds right part of the federated join.

First, we notice that in both cases, Semagrow returns the same number of results. This is an indication that the optimization technique is correct, i.e., the additional filter does not prune any shapes that are within the given distance.

Regarding semagrow-std, which issues the unoptimized query in INVEKOS, we notice that every query processing requires at least 57 seconds to find all shapes that are within the specific distance, even if the distance is small (i.e., 10 meters). This is an expected behavior, because the federated source requires to compare the distance of the every shape in the dataset from the LUCAS WKT. For larger distances we have even higher query processing times. This fact can be explained because

Table 4.4: Experimental results for Q4.

query	distance	semagrow-std query proc. time	semagrow-std #results	semagrow-opt query proc. time	semagrow-opt shapes pruned by optimization		semagrow-opt #results
Q4	10 m	58 sec	2	0.1 sec	2,008,134	(99.99%)	2
Q4	100 m	57 sec	7	0.1 sec	2,008,129	(99.99%)	7
Q4	1 km	58 sec	70	0.1 sec	2,008,004	(99.99%)	70
Q4	10 km	57 sec	4,739	1.2 sec	1,996,169	(99%)	4,739
Q4	50 km	72 sec	141,973	26 sec	1,702,032	(84%)	141,973
Q4	100 km	110 sec	528,026	86 sec	1,212,393	(60%)	528,026

of the larger result set of queries for distance greater than 50 km. As a result, we can conclude that we have to wait at least 1 minute to retrieve the shapes that are within a specific distance, regardless of the length of the distance.

Regarding semagrow-opt, which issues the optimized query in INVEKOS, we notice that the query processing time is analogous to the size of the distance, i.e., for a smaller distance we have a faster query processing time. This behavior can be explained due to the additional filter placed by the optimization, because there exists a connection with the amount of the pruning of the additional filter with the query processing time. Due to the pruning of the extra filter, the federated source has less candidate shapes to compare their distance from the LUCAS WKT. Moreover, since the `geof:sfIntersects` function of the additional filter can be evaluated using the spatial index, this filter does not introduce any time overhead in the evaluation of the query.

Comparing the two approaches, we observe that the optimized version reduces the query processing time by 3 orders of magnitude for distances less than 1 km and by 2 orders of magnitude for distances around 10 km. For larger distances, the time difference is less pronounced, but in any case, we can safely conclude that the optimization technique can be effective for federated within-distance queries for any distance length.

4.5 Related Work

The literature on federated geospatial query processing is very sparse, in either the Semantic Web community, the geographical information systems community, or the wider databases community. Recent studies [3, Section 5] find that there is no mature federated GeoSPARQL query processing system. A recent survey that investigates federated query processing and other data integration methods only gives the static transformation of distributed data into a single store as a means of integrating geospatial data [17, Section 4].

PostGIS has already a build-in function calculating the geometries that are within a given distance, called `ST_DWithin` [27], that GeoSPARQL does not support. `ST_DWithin` function includes a bounding box comparison that makes use of any indexes that are available on the geometries, which is what we are implementing with our geospatial join optimization method.

Similarly, this is kind of how PostGIS proposes to handle the nearest neighbour search [23]. Traditionally, the naive way to find the nearest neighbour is to force the database to calculate the distance between the query geometry and every candidate geometry and then sort them all. For a large table of candidate geometries, it is not a efficient approach. One way to improve performance is to add an index constraint to the search; first find which candidate geometries overlap or touch with an arbitrary box around the query geometry and only calculate the actual distance with them. The above introduces the problem of somehow choosing the smallest box, and that is why PostGIS introduces KNN, a pure index based nearest neighbour search. PostGIS uses an R-Tree

index implemented on top of GiST (Generalized Search Tree) to index GIS data. The KNN system evaluates distances between bounding boxes inside the PostGIS GiST index. So, in the case of within-distance queries the computations will be between the bounding boxes of geometries; they will not be precise on the exact geometries.

Another optimization can be performed by creating on-the-fly spatial indexing for spatial objects [31]. The queries will be more efficient due to the indexing being dependent specifically on the spatial predicate and the geometry type of each spatial object. Geospatial joins can also be optimized by rewriting the GeoSPARQL queries into simpler more primitive sub-queries [31] and parallelizing them [30]. For example, a query using the nearby function can be divided to hasGeometry and within-distance (or distance) sub-queries. Alternatively, the filtering within-distance computation can be parallelized by partitioning the spatial objects into approximately equal-sized patches based on spatial indices. The spatial sub-queries within each patch and around the borders of the patches are then processed in parallel, and their results are combined.

4.6 Conclusion

We presented a geospatial join optimization method for federated within-distance queries. In this optimization, the query execution module of a federation engine can be extended, so that it rewrites the query to be issued to the geospatial source endpoint by adding an extra filter to the within-distance filter. This can help the federated source to calculate faster the result set since the additional filter can make use of the spatial index of the source. The implementation of our method is provided as open source, integrated with the Semagrow federated GeoSPARQL processor.

In the experimental setup, we showed that the optimization can speed up the queries extremely, for distances up to 1 km. This behavior can be proved useful for real-world applications and use cases of the project, such as the data validation of crop type information. Apart from this, our technique can reduce the query processing time for larger distances as well.

Finally, we have to stress the usefulness of the the query set for the LUCAS-INVEKOS data validation task, which was compiled with a nice collaboration with the UNITN partner of Extreme Earth. Apart from obtained from an actual, practical application of geospatial and linked data querying, the queries use GeoSPARQL constructs that challenge all phases of federated query processing, from source selection to query planing and execution, apart from the within-distance operation [28].

5. Implementation

In this chapter, we will focus on the implementation of the new version of Semagrow, which is the software resulting from Task T3.4. Apart from the methodological extensions presented and evaluated in the previous chapters, we also extended Semagrow with other development work on source selection (cf. Section 5.1), query planning (cf. Section 5.2), query execution (cf. Section 5.3), and integration with other systems in the context of Extreme Earth (cf. Section 5.4).

5.1 Work on Source Selection

In this section we report the development work on the source selector of Semagrow. We have extended the pre-Extreme-Earth source selection of Semagrow with a thematic source selection and a geospatial source selector.

Thematic Source Selection

Properties of standard vocabularies, which can appear possibly in all sources of a federation, present a challenge in the efficient evaluation of a query. When evaluating a triple pattern that contains a property such as `rdf:type`, the source selector is prone to overestimate the set of relevant sources, thus increasing both network traffic and the overall query processing time. In geospatial data this phenomenon is more pronounced; a feature `?x` is linked with the WKT of its geometry `?wkt` using *chains of known properties* of the form `?x geo:hasGeometry ?g . ?g geo:asWKT ?wkt`, where all members of the chain almost always appear in the same dataset. An overestimation of the set of sources for such triple patterns would lead to fetching redundant bindings for the variable in the middle of the chain.

Another characteristic of real-world datasets is the fact that, quite often, the URIs that are included in a dataset have a common unique prefix (for example, all DBpedia entity URIs start with `http://dbpedia.org/resource/`). Thus, we can exploit this information through a URI-prefix-based source selector in order to prune the set of sources for chains of triple patterns that link a feature with its WKT. For instance consider the following subquery:

```
?x lucas:hasLC1 ?lc1 . ?x geo:hasGeometry ?g . ?g geo:asWKT ?wkt
```

If the property `lucas:hasC1` appears in a single source of the federation and all URIs of this source have a common unique prefix, we can conclude that the remaining properties belong only to this source by using a URI-prefix *join-aware* source selection method. Pre-Extreme-Earth Semagrow didn't use such a method.

For this reason, we extended the thematic source selector of Semagrow with a URI-prefix *join-aware* source selector similar to that of HiBISCuS [24], which prunes non-joinable sources based on the URI prefixes of the shared variables. This source selector uses URI-prefix metadata for each source and are expressed using the Sevod vocabulary [11].

The pull requests that refer to this extension of Semagrow can be found at
`https://github.com/semagrow/semagrow/pull/37`
`https://github.com/semagrow/semagrow/pull/75`
`https://github.com/semagrow/semagrow/pull/76`

Geospatial Source Selection

The next development of the source selection component of Semagrow is the geospatial selection method, which was introduced and analyzed in Chapter 3.

The pull requests that refer to this extension of Semagrow can be found at
`https://github.com/semagrow/semagrow/pull/85`
`https://github.com/semagrow/semagrow/pull/44`

5.2 Work on Query Planning

In this section we report the development work on the query planning and optimization of Semagrow. We have extended the pre-Extreme-Earth query planner to support complex queries and with some pushdown optimizations.

Support of Complex Queries

During Extreme Earth, we have faced with the need of executing complex SPARQL queries, in the context of the LUCAS-INVEKOS data validation task (cf. Section 4.2)). The corresponding SPARQL queries have a complex nature that challenges query planning. In particular, the queries use the following complex characteristics:

- the use a *subquery* for discovering the *closest* INVEKOS instance, i.e., the use of an inner SPARQL query, together with an `ORDER` operation over the result of a function (that is `geof:distance`) that appears in a `BIND` expression, and a `LIMIT 1` operation.
- the use *negation*, in the form of the `FILTER NOT EXISTS` operator to check that there does not exist any matching INVEKOS instance.

These queries are more complex than the usual queries found in the federated SPARQL querying domain, because they are not consisting simply of joins between triple patterns and `FILTER` operations, but also make use of the `ORDER`, `LIMIT`, `BIND`, and `NOT EXISTS` operators. The queryset appears in Section A.2 of the Appendix.

In order to support these queries, we reworked some parts of the query model of Semagrow. The pull requests that refer to this work can be found at
`https://github.com/semagrow/semagrow/pull/63`
`https://github.com/semagrow/semagrow/pull/77`

Filter Pushdown Optimization

As discussed in the previous chapters, filter pushdown is an important optimization in the context of geospatial data, because pushing the geospatial filters in the geospatial sources can improve the query processing time not only due to the reduction of the redundant intermediate results, but also because the federated geospatial maintain spatial indexes which speed-up the evaluation of geospatial filters.

Given a SPARQL expression A and a source endpoint s, we use the notation A @ s to express that the expression A is evaluated in the endpoint s.

Pre-Extreme-Earth Semagrow uses the following filter pushdown optimization: Let A and B be SPARQL expressions, s be a source endpoint, and C be a SPARQL expression such that all free variables of C appears only either on A or on B. Then, the expression

$$\big(A \texttt{ BIND\_JOIN } (B \texttt{ @ } s)\big) \texttt{ FILTER } C$$

is transformed into

$$A \texttt{ BIND\_JOIN } \big((B \texttt{ FILTER } C) \texttt{ @ } s\big)$$

We have extended this filter pushdown optimizer as follows: Let A and B be SPARQL expressions, $s_1, s_2, \ldots s_n$ be source endpoints, and C be a SPARQL expression such that all free variables of C appears only either on A or on B. Then, the expression

$$\Big(A \texttt{ BIND\_JOIN } \big((B \texttt{ @ } s_1) \texttt{ UNION } (B \texttt{ @ } s_2) \texttt{ UNION } \cdots \texttt{ UNION } (B \texttt{ @ } s_n)\big)\Big) \texttt{ FILTER } C$$

is transformed into

$$A \texttt{ BIND\_JOIN } \Big(\big((B \texttt{ FILTER } C) \texttt{ @ } s_1\big) \texttt{ UNION } \big((B \texttt{ FILTER } C) \texttt{ @ } s_2\big) \texttt{ UNION } \cdots \\ \cdots \texttt{ UNION } \big((B \texttt{ FILTER } C) \texttt{ @ } s_n\big)\Big)$$

This optimization helps us to push the corresponding filters in situations where a subquery with geospatial filters (for instance `?s geo:asWKT ?w . FILTER(geof:sfWithin(?w "KNOWN_WKT"))`) has to be evaluated in more than one geospatial sources of the federation.

The pull request that refers to this extension of Semagrow can be found at
`https://github.com/semagrow/semagrow/pull/44`

Join Pushdown Optimization over Disjoint Sources

Pre-Extreme-Earth version of Semagrow used an exclusive group optimization; that is, all triple patterns that appear in a single source are "grouped together" in a same subquery, in order to avoid fetching unnecessary data. Exclusive group optimization is actually a join pushdown optimization. In particular: Let A and B be SPARQL expressions, and s be a source endpoint. Then, the expression

$$(A \texttt{ @ } s) \texttt{ BIND\_JOIN } (B \texttt{ @ } s)$$

is transformed into

$$(A \texttt{ . } B) \texttt{ @ } s$$

Continuing this line of thought, we observe that such a grouping can be applied also in cases where the A and B expressions are to be evaluated in a *set of disjoined sources*. Thus, we have extended Semagrow with a join pushdown optimization over unions of disjoint experssions.

Let A and B be SPARQL expressions, and $s_1, s_2, \ldots s_n$ be source endpoints. Then, the expression

$$\big((A \texttt{ @ } s_1) \texttt{ UNION } (A \texttt{ @ } s_2) \texttt{ UNION } \cdots \texttt{ UNION } (A \texttt{ @ } s_n)\big) \\ \texttt{BIND\_JOIN} \\ \big((B \texttt{ @ } s_1) \texttt{ UNION } (B \texttt{ @ } s_2) \texttt{ UNION } \cdots \texttt{ UNION } (B \texttt{ @ } s_n)\big)$$

is transformed into

$$\big((A \texttt{ . } B) \texttt{ @ } s_1\big) \texttt{ UNION } \big((A \texttt{ . } B) \texttt{ @ } s_2\big) \texttt{ UNION } \cdots \texttt{ UNION } \big((A \texttt{ . } B) \texttt{ @ } s_n\big)$$

provided that $\big((A \texttt{ @ } s_i) \texttt{ BIND\_JOIN } (B \texttt{ @ } s_j)\big) = \emptyset$ for all $i \neq j$.

This optimization can be effective if the process of checking that the expressions that appear in different endpoints are disjoint is pretty fast. With Semagrow, which uses Sevod metadata [11], it is possible to distinguish such cases without issuing queries in the endpoints. In our implementation, we consider these two cases (which cover the majority of the queries during Extreme Earth):

Case 1: All triples between s_1 and s_2 cannot be joined, or equivalently, all joins between s_1 and s_2 have selectivity value equal to 0. This can be expressed in Sevod as follows:

```
[] a svd:Join ;
  svd:joins [ void:sparqlEndpoint e1 ];
  svd:joins [ void:sparqlEndpoint e2 ];
  svd:selectivity [ svd:selectivityValue 0 ] .
```

where `e1`, `e2` are the SPARQL endpoints of s_1, s_2 respectively. In this case, if all common variables of A and B appear only in triple patterns, then we can conlude that A @ s_1 and B @ s_2 are non-joinable.

Case 2: Every WKT literal that appears in s_1 is disjoint from any WKT literal that appears in s_2. Sevod allows us for each source s to define a polygon that *contains* every spatial object that appears in s. This can be expressed in Sevod as follows:

```
[] a void:Dataset ; void:sparqlEndpoint e1; svd:boundingWKT wkt1 .
[] a void:Dataset ; void:sparqlEndpoint e2; svd:boundingWKT wkt2 .
```

where `e1`, `e2` are the SPARQL endpoints of s_1, s_2 respectively. In this case, if the WKT literal `wkt1` is disjoint from the WKT literal `wkt2`, and if the common variable between A and B appears in a geospatial filter that expresses a topological relation other than *disjoint*, then we can conlude that A @ s_1 and B @ s_2 are non-joinable.

This optimization helps us to group both thematic and geospatial triple patterns that refer to the same feature in the same subquery (for instance `?s geo:hasGeometry ?g . ?g geo:asWKT ?w . FILTER(geof:sfWithin(?w "KNOWN_WKT"))`), if many geospatial sources of the federation contain relevant data, provided that the data is partitioned into several disjoint sources (i.e., such that each source refers to a specific grid partition or a specific administrative region, or area of responsibility).

The pull request that refers to this extension of Semagrow can be found at
`https://github.com/semagrow/semagrow/pull/88`

5.3 Work on Query Execution

In this section we report the development work on the query execution component of Semagrow. We have extended the pre-Extreme-Earth query execution mechanism with evaluators of geospatial functions, an optimized query executor for within-distance geospatial joins, and a query executor for connecting directly with PostGIS databases.

Geospatial function evaluation

The experiments we have conducted during the project, indicated that filters with geospatial functions should be pushed to the federated geospatial sources whenever possible. However, in case

Table 5.1: List of supported GeoSPARQL and stSPARQL functions.

language	family	functions
GeoSPARQL	non-topological	`geof:distance`, `geof:boundary`, `geof:buffer`, `geof:convexHull`, `geof:difference`, `geof:envelope`, `geof:intersection`, `geof:getSRID`, `geof:union`, `geof:symDifference`, `geof:relate`.
	simple feature	`geof:sfEquals`, `geof:sfDisjoint`, `geof:sfTouches`, `geof:sfCrosses`, `geof:sfIntersects`, `geof:sfWithin`, `geof:sfContains`, `geof:sfOverlaps`.
	Egenhofer	`geof:ehEquals`, `geof:ehDisjoint`, `geof:ehOverlap`, `geof:ehContains`, `geof:ehCovers`, `geof:ehCoveredBy`, `geof:ehInside`, `geof:ehMeet`.
	RCC8	`geof:rcc8eq`, `geof:rcc8dc`, `geof:rcc8ec`, `geof:rcc8tppi`, `geof:rcc8ntpp`, `geof:rcc8ntppi`, `geof:rcc8po`, `geof:rcc8tpp`.
stSPARQL	basic	`strdf:dimension`, `strdf:isEmpty`, `strdf:srid`, `strdf:asText`, `strdf:asGML`, `strdf:geometryType`, `strdf:isSimple`.
	topological	`strdf:equals`, `strdf:disjoint`, `strdf:intersects`, `strdf:touches`, `strdf:crosses`, `strdf:within`, `strdf:contains`, `strdf:overlaps`, `strdf:relate`.
	topological (min. bounding boxes)	`strdf:mbbEquals`, `strdf:mbbWithin`, `strdf:mbbIntersects`, `strdf:mbbContains`.
	directional	`strdf:left`, `strdf:right`, `strdf:above`, `strdf:below`.
	spatial analysis	`strdf:buffer`, `strdf:distance`, `strdf:boundary`, `strdf:extent`, `strdf:envelope`, `strdf:convexHull`, `strdf:intersection`, `strdf:union`, `strdf:difference`, `strdf:symDifference`, `strdf:area`.

when the query planner of Semagrow cannot push the geospatial functions in the federated sources, Semagrow is required to provide an implementation of all geospatial functions of GeoSPARQL and stSPARQL so that it can join and filter the partial results received from each data source.

To this end, we first leveraged the GeoSPARQL operators implemented in the RDF4J 2.5.5 library[1], mapping them to their stSPARQL equivalents to cover both GeoSPARQL and (the geospatial part of) stSPARQL. However, some stSPARQL functions, such as `strdf:area`, do not have a GeoSPARQL equivalent and are (thus) not supported by RDF4J. In these cases we provide an implementation using the Java Topology Suite (JTS) library[2]. The supported GeoSPARQL stSPARQL functions are shown in Table 5.1.

The pull request that refers to this extension of Semagrow can be found at
`https://github.com/semagrow/semagrow/pull/36`

Geospatial Join Optimization for within-distance queries

The next development of the query execution component of Semagrow is the development of an optimized query executor that contains the geospatial join optimization method for within-distance queries, which was introduced and analyzed in Chapter 4.

[1] Cf. `https://rdf4j.org/documentation/programming/geosparql/`
[2] Cf. `https://locationtech.github.io/jts/`

The pull request that refers to this extension of Semagrow can be found at
`https://github.com/semagrow/semagrow/pull/87`

PostGIS Connector

Pre-Extreme-Earth Semagrow allowed us to define query executor plugins for integrating data sources other than SPARQL endpoints. In order to experiment with geospatial data sources other than GeoSPARQL endpoints during Extreme Earth, we developed a query executor for PostGIS databases.

We consider PostGIS databases that contain only shapes without any thematic information. In particular, we assume that each database has a table named `geometries` with at least two columns. One column named `id` (which is of type integer) which is the shape identifier, and one named `wkt` (which is of type geometry), which encodes to the geometric representation of the shape.

The RDF mapping of database data of a database named `DBNAME` we use can be represented with the following R2RML mapping [5]:

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix pgm: <http://rdf.semagrow.org/pgm/DBNAME/>.

<#TriplesMap1>
    rr:logicalTable [ rr:tableName "geometries" ];
    rr:subjectMap [
        rr:template "http://rdf.semagrow.org/pgm/DBNAME/resource/{id}";
        rr:class pgm:geometry;
    ];
    rr:predicateObjectMap [
        rr:predicate geo:asWKT;
        rr:objectMap [ rr:column "wkt" ];
    ].
```

For example, the following table

geometries

id	wkt
33	POINT(1 2)
19	POINT(3 4)

would be equivalent with this set of triples

```
<http://rdf.semagrow.org/pgm/shp/resource/33>  rdf:type
      <http://rdf.semagrow.org/pgm/shp/geometry> .
<http://rdf.semagrow.org/pgm/shp/resource/33>  geo:asWKT
      "<http://www.opengis.net/def/crs/EPSG/0/4326> POINT(1 2)"^^geo:wktLiteral .
<http://rdf.semagrow.org/pgm/shp/resource/19>  rdf:type
      <http://rdf.semagrow.org/pgm/shp/geometry> .
<http://rdf.semagrow.org/pgm/shp/resource/19>  geo:asWKT
      "<http://www.opengis.net/def/crs/EPSG/0/4326> POINT(3 4)"^^geo:wktLiteral .
```

provided that the database name is `shp`.

The PostGIS connector operates by translating a SPARQL subquery into SQL for retrieving the data from the remote database. It supports all GeoSPARQL simple feature functions, as well as the GeoSPARQL `geof:distance` non-topological function. For example, a SPARQL query like the one below

```
SELECT * WHERE {
  ?s rdf:type pgm1:geometry . ?s geo:asWKT ?wkt .
  FILTER( geof:sfWithin(?wkt, "POLY_WKT") )
}
```

is translated to the following SQL query

```
SELECT table0.id AS s, ST_AsText(table0.wkt) AS wkt
FROM geometries table0
WHERE ST_Within( ST_GeomFromText(ST_AsText(table0.wkt), 4326),
                 ST_GeomFromText(ST_AsText('POLY_WKT'), 4326)  )
```

After issuing the SQL query in the remote database, the PostGIS connector retrieves database query results and transforms them into valid SPARQL query results. Semagrow allows federating multiple PostGIS databases provided that they have different name. Moreover, the connector supports bind join with multiple bindings over the federated PostGIS databases. For instance, in queries like:

```
PREFIX pgm1: <http://rdf.semagrow.org/pgm/shp1/>
PREFIX pgm2: <http://rdf.semagrow.org/pgm/shp2/>

SELECT * WHERE {
  ?s1 rdf:type pgm1:geometry . ?s1 geo:asWKT ?wkt1 .
  ?s2 rdf:type pgm2:geometry . ?s2 geo:asWKT ?wkt2 .
  FILTER( geof:distance(?wkt1,?wkt2,uom:metre) < DIST )
}
```

where we have more than one results for `?s1` and `?wkt1` (from the `shp1` database), the connector can send one SQL query to the `shp2` database with the previous bindings. The connector in such cases uses UNION in the query translation.

In order for the PostGIS connector to be used (instead of the original SPARQL connector) the endpoint in the Semagrow metadata should have the form `<postgis://ip:port/dbname>`, where `ip` is the database IP, `port` is the database port, and `dbname` is the database name.

The pull request that refers to this extension of Semagrow can be found at
`https://github.com/semagrow/semagrow/pull/89`

5.4 Integration with other systems

Finally, we report the development work that we performed for integrating Semagrow with other components of Extreme Earth, i.e., the KOBE Benchmarking Engine (developed in Task T3.5, will be presented in Deliverable D3.5 in M36 of the project) and Hopsworks (cf. Deliverable D1.6 [8]).

Integration with KOBE

The KOBE Open Benchmarking Engine [12] is a system that provides an extensible platform for benchmarking federated query processors. Among many features, it provides a WebUI where a set of experimental metrics are visualized using 3 Kibana dashboards. We distinguish two kinds of metrics: those calculated by the client side and those calculated by the federation engine itself. In order for the metrics to be visualized, a federation should be extended with some KOBE-specific log lines.

In order to provide full support for evaluation metrics, we did the following:

1. We extended the Semagrow query string parser to parse the first line of the query according to a KOBE-specific regex pattern that contains the signature of the query.
2. We extended Semagrow with some code that calculates the evaluation metrics that are to be provided by the federation side, which are the following:
   - Source selection time
   - Query planning time
   - Query execution time
   - Number of sources contained in the execution plan
3. We added some KOBE-specific log lines that contain the aformentioned metrics and the query signature.

The pull request that refers to this extension of Semagrow can be found at
`https://github.com/semagrow/semagrow/pull/52`

Integration with Hopsworks

Semagrow is integrated in the Hopsworks platform. In the following, we provide the instructions for installing Semagrow on Hops' JBoss application server. The installation can also be performed using the automated procedures provided by Hops for the deployment of the modules.

As prerequisites, we assume that you have already prepared the WAR file for the Semagrow deployment, and its two configuration files, namely `repository.ttl` and `metadata.ttl`.

1. In order to obtain the WAR file of Semagrow, clone the Semagrow GitHub repository `https://github.com/semagrow/semagrow` and build it using Maven. The WAR file of Semagrow should appear in `semagrow/http-endpoint/target/SemaGrow.war` (for more detailed instructions on how to build Semagrow, check the README of the GitHub repository).
2. Regarding the configuration files, samples of `repository.ttl` and `metadata.ttl` can be found as resources of the `http` module of Semagrow. Moreover, a tool that helps with the creation of Semagrow metadata can be found in the repository `https://github.com/semagrow/sevod-scraper`.

For installing Semagrow on Hopsworks, in the machine where you have Hopsworks deployed, you should do the following:

1. Create the directory `/etc/default/semagrow` and place there the configuration files of Semagrow, namely `repository.ttl` and `metadata.ttl`

2. Place the Semagrow WAR into the `/srv/hops/domains/domain1/autodeploy` directory
3. Then, Semagrow is ready to be accessed from `http://DOMAIN/SemaGrow/index.jsp`

The integration of Semagrow with Hopsworks, required some work for supporting the deployment of Semagrow on Hops' Glassfish server, which can be found at
`https://github.com/semagrow/semagrow/pull/38`

6. Conclusion

In this deliverable we presented the new version of the Semagrow query federation engine, which was developed during the Task T3.4 of the Extreme Earth project. The resulting method and software provides unified access of linked geospatial data from multiple, possibly heterogeneous, geospatial data servers. All phases of the federated query processing (namely source selection, query planning and query execution) of Semagrow were extended in order to federate geospatial linked data, and Semagrow has been successfully integrated within Hopsworks. In addition, we introduce two novel techniques for federating geospatial linked data; a geospatial source selector and a federated geospatial join optimization, which have been both evaluated on data and queries from the use cases of the Extreme Earth project; both experiments show that each of the techniques can substantially improve query processing time.

Regarding future work in the context of ExtremeEarth, we plan to conduct an extensive end-to-end evaluation of the new version of Semagrow, using the KOBE Open Benchmarking Engine, which is the benchmarking framework developed in Task T3.5 of the project. In these experiments, we will combine the cluster-level scalability offered by Strabo2 with Semagrow's ability to transparently federate multiple such clusters. The aim is to prove that the combination of these key Extreme Earth technologies can bring geospatial linked data query processing to the order of magnitude of petabytes.

A. Queries

In this appendix we present the GeoSPARQL queries that we have used in our experiments, i.e, the queries used in the geospatial source selection experiment (cf. Section 3.4) and the ones used in the geospatial join optimization experiment (cf. Section 4.4).

In all queries, we assume the following set of prefixes:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>

PREFIX gadm: <http://gadm.org/ontology#>
PREFIX fs: <http://earthanalytics.eu/fs/ontology/>
PREFIX invekos: <http://deg.iit.demokritos.gr/invekos/>
PREFIX lucas: <http://deg.iit.demokritos.gr/lucas/>
PREFIX lucas_r: <http://deg.iit.demokritos.gr/lucas/resource/>
PREFIX lictm: <http://deg.iit.demokritos.gr/lictm/>
```

A.1 Geospatial Source Selection experiment queries

Q1. Municipalities intersecting a given WKT

```
SELECT * WHERE {
  ?s rdf:type gadm:AdministrativeUnit3.
  ?s gadm:has_NAME_1 ?adm1 .
  ?s gadm:has_NAME_2 ?adm2 .
  ?s gadm:has_NAME_3 ?adm3.
  ?s geo:hasGeometry ?g .
  ?g geo:asWKT ?wkt .

  FILTER (geof:sfIntersects(?wkt, "POLYGON ((15.11 48.68, 15.11 48.72, 15.19
    48.72, 15.19 48.68, 15.11 48.68))"^^geo:wktLiteral)).
}
```

Q2. Municipalities intersecting another given WKT

```
SELECT * WHERE {
  ?s rdf:type gadm:AdministrativeUnit3 .
  ?s gadm:has_NAME_1 ?adm1 .
  ?s gadm:has_NAME_2 ?adm2 .
  ?s gadm:has_NAME_3 ?adm3.
  ?s geo:hasGeometry ?g .
  ?g geo:asWKT ?wkt .

  FILTER (geof:sfIntersects(?wkt, "POLYGON ((14.87 48.18, 14.87 48.14, 14.94
    48.14, 14.94 48.18, 14.87 48.18))"^^geo:wktLiteral)).
}
```

Q3. Snow cover within 5 km from a given municipality

```
SELECT * WHERE {
  ?u rdf:type gadm:AdministrativeUnit3 .
  ?u gadm:has_NAME_3 "Deutsch Kaltenbrunn" .
  ?u geo:hasGeometry ?x .
  ?x geo:asWKT ?wkt1 .

  ?s rdf:type fs:FoodSecurityObservation .
  ?s fs:hasDN ?dn .
  ?s fs:hasRECDATE "2018-02-10T00:00:00"^^xsd:dateTime .
  ?s geo:hasGeometry ?y .
  ?y geo:asWKT ?wkt2 .

  FILTER (geof:distance(?wkt1, ?wkt2, uom:metre) < 5000) .
}
```

Q4. Potato fields within a given municipality

```
SELECT * WHERE {
  ?u rdf:type gadm:AdministrativeUnit3 .
  ?u gadm:has_NAME_3 "Kirchberg am Walde" .
  ?u geo:hasGeometry ?x .
  ?x geo:asWKT ?wkt1 .

  ?f invekos:hasID ?id .
  ?f invekos:hasCropTypeName "POTATO" .
  ?f invekos:hasArea ?area .
  ?f geo:hasGeometry ?y .
  ?y geo:asWKT ?wkt3 .

  FILTER (geof:sfWithin(?wkt3, ?wkt1)).
}
```

Q5. Snow-covered potato fields intersecting a given WKT

```
SELECT * WHERE {
  ?f invekos:hasID ?id .
  ?f invekos:hasCropTypeName "POTATO" .
  ?f invekos:hasArea ?area .
  ?f geo:hasGeometry ?x .
  ?x geo:asWKT ?wkt3 .

  ?s rdf:type fs:FoodSecurityObservation .
  ?s fs:hasDN ?dn .
  ?s fs:hasRECDATE "2018-02-10T00:00:00"^^xsd:dateTime .
  ?s geo:hasGeometry ?y .
  ?y geo:asWKT ?wkt2 .
  FILTER (0 < ?dn && ?dn < 50) .

  FILTER (geof:sfIntersects(?wkt3, ?wkt2)).
  FILTER (geof:sfIntersects(?wkt3, "POLYGON ((15.11 48.68, 15.11 48.72, 15.19
    48.72, 15.19 48.68, 15.11 48.68))"^^geo:wktLiteral)).
```

```
  FILTER (geof:sfIntersects(?wkt2, "POLYGON ((15.11 48.68, 15.11 48.72, 15.19
    48.72, 15.19 48.68, 15.11 48.68))"^^geo:wktLiteral)).
}
```

Q6. Snow-covered potato fields within a given municipality

```
SELECT * WHERE {
  ?u rdf:type gadm:AdministrativeUnit3 .
  ?u gadm:has_NAME_3 "Kirchberg am Walde" .
  ?u geo:hasGeometry ?x .
  ?x geo:asWKT ?wkt1 .

  ?f invekos:hasID ?id .
  ?f invekos:hasCropTypeName "POTATO" .
  ?f invekos:hasArea ?area .
  ?f geo:hasGeometry ?y .
  ?y geo:asWKT ?wkt2 .

  ?s rdf:type fs:FoodSecurityObservation .
  ?s fs:hasDN ?dn .
  ?s fs:hasRECDATE "2018-02-10T00:00:00"^^xsd:dateTime .
  ?s geo:hasGeometry ?z .
  ?z geo:asWKT ?wkt3 .
  FILTER (0 < ?dn && ?dn < 50) .

  FILTER (geof:sfWithin(?wkt2, ?wkt1)).
  FILTER (geof:sfWithin(?wkt2, ?wkt3)).
  FILTER (geof:sfIntersects(?wkt1, ?wkt3)).
}
```

Q7. Potato fields within 5km from snow cover and intersecting a given WKT

```
SELECT * WHERE {
  ?f invekos:hasID ?id .
  ?f invekos:hasCropTypeName "POTATO" .
  ?f invekos:hasArea ?area .
  ?f geo:hasGeometry ?x .
  ?x geo:asWKT ?wkt2 .

  ?s rdf:type fs:FoodSecurityObservation .
  ?s fs:hasDN ?dn .
  ?s fs:hasRECDATE "2018-02-10T00:00:00"^^xsd:dateTime .
  ?s geo:hasGeometry ?y .
  ?y geo:asWKT ?wkt3 .
  FILTER (0 < ?dn && ?dn < 50) .

  FILTER (geof:distance(?wkt2, ?wkt3, uom:metre) < 5000).
  FILTER (geof:sfIntersects(?wkt2, "POLYGON ((15.11 48.68, 15.11 48.72, 15.19
    48.72, 15.19 48.68, 15.11 48.68))"^^geo:wktLiteral)).
  FILTER (geof:sfIntersects(?wkt3, "POLYGON ((15.11 48.68, 15.11 48.72, 15.19
    48.72, 15.19 48.68, 15.11 48.68))"^^geo:wktLiteral)).
}
```

Q8. Potato fields within 5km from snow cover and within a given municipality

```
SELECT * WHERE {
  ?u rdf:type gadm:AdministrativeUnit3 .
  ?u gadm:has_NAME_3 "Deutsch Kaltenbrunn" .
  ?u geo:hasGeometry ?x .
  ?x geo:asWKT ?wkt1 .

  ?f invekos:hasID ?id .
  ?f invekos:hasCropTypeName "POTATO" .
  ?f invekos:hasArea ?area .
  ?f geo:hasGeometry ?y .
  ?y geo:asWKT ?wkt2 .

  ?s rdf:type fs:FoodSecurityObservation .
  ?s fs:hasDN ?dn .
  ?s fs:hasRECDATE "2018-02-10T00:00:00"^^xsd:dateTime .
  ?s geo:hasGeometry ?z .
  ?z geo:asWKT ?wkt3 .
  FILTER (0 < ?dn && ?dn < 60) .

  FILTER (geof:sfWithin(?wkt2, ?wkt1)).
  FILTER (geof:distance(?wkt2, ?wkt3, uom:metre) < 5000).
  FILTER (geof:sfIntersects(?wkt1, ?wkt3)).
}
```

A.2 Geospatial Join Optimization experiment queries

Q1: nearest INVEKOS instance if it is within 10m and their crop types match

```
SELECT * WHERE {
  {
    SELECT * WHERE {
      lucas_r:$(ID) geo:hasGeometry ?l_geom_id .
      ?l_geom_id geo:asWKT ?l_geom .
      ?inv invekos:hasCropTypeNumber ?i_ctype .
      ?inv geo:hasGeometry ?i_geom_id .
      ?i_geom_id geo:asWKT ?i_geom .
      BIND(geof:distance(?l_geom, ?i_geom, uom:metre) as ?distance) .
      FILTER(?distance < 10) .
    }
    ORDER BY ASC(?distance)
    LIMIT 1
  }
  lucas_r:$(ID) lucas:hasLC1 ?lc1 .
  lucas_r:$(ID) lucas:hasLC1_SPEC ?lc1_sp .
  ?c lictm:lucasLC1 ?lc1 .
  ?c lictm:lucasLC1_spec ?lc1_sp .
  ?c lictm:invekosCropTypeNumber ?l_ctype .
  FILTER(?l_ctype = ?i_ctype) .
}
```

`$(ID)` takes 2488 values from the set $[1, 8840]$.

Q2: nearest INVEKOS instance if it is within 10m and their crop types do not match

```
SELECT * WHERE {
  {
    SELECT * WHERE {
      lucas_r:$(ID) geo:hasGeometry ?l_geom_id .
      ?l_geom_id geo:asWKT ?l_geom .
      ?inv invekos:hasCropTypeNumber ?i_ctype .
      ?inv geo:hasGeometry ?i_geom_id .
      ?i_geom_id geo:asWKT ?i_geom .
      BIND(geof:distance(?l_geom, ?i_geom, uom:metre) as ?distance) .
      FILTER(?distance < 10) .
    }
    ORDER BY ASC(?distance)
    LIMIT 1
  }
  FILTER NOT EXISTS {
    lucas_r:$(ID) lucas:hasLC1 ?lc1 .
    lucas_r:$(ID) lucas:hasLC1_SPEC ?lc1_sp .
    ?c lictm:lucasLC1 ?lc1 .
    ?c lictm:lucasLC1_spec ?lc1_sp .
    ?c lictm:invekosCropTypeNumber ?l_ctype .
    FILTER(?l_ctype = ?i_ctype) .
  }
}
```

`$(ID)` takes 2488 values from the set $[1, 8840]$.

Q3: LUCAS instance if there is no INVEKOS instance within 10m

```
SELECT * WHERE {
  lucas_r:$(ID) lucas:hasLC1 ?lc1 .
  FILTER NOT EXISTS {
    lucas_r:$(ID) geo:hasGeometry ?l_geom_id .
    ?l_geom_id geo:asWKT ?l_geom .
    ?inv invekos:hasCropTypeNumber ?i_ctype .
    ?inv geo:hasGeometry ?i_geom_id .
    ?i_geom_id geo:asWKT ?i_geom .
    FILTER (geof:distance(?l_geom, ?i_geom, uom:metre) < 10) .
  }
}
```

`$(ID)` takes 2488 values from the set $[1, 8840]$.

Q4: all INVEKOS instances within D meters from a specific LUCAS instance

```
SELECT * WHERE {
  lucas_r:$(ID) geo:hasGeometry ?l .
  ?l geo:asWKT ?wkt1 .
  ?i invekos:hasID ?id .
  ?i geo:hasGeometry ?g .
```

```
  ?g geo:asWKT ?wkt2 .

  FILTER( geof:distance(?wkt1, ?wkt2, uom:metre) < $(DIST) ) .
}
```

`$(DIST)` is one of 10, 100, 1000, 10000, 50000, 100000

Bibliography

[1] Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2011.

[2] Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. Describing linked datasets with the VoID vocabulary. W3C Interest Group Note, 3 March 2011, March 2011.

[3] Konstantina Bereta, Hervé Caumont, Ulrike Daniels, Daems Dirk, Erwin Goor, Manolis Koubarakis, Despina-Athanasia Pantazi, George Stamoulis, Sam Ubels, Valentijn Venus, and Firman Wahyudi. The Copernicus App Lab project: Easy access to Copernicus data. In *Proceedings of the 22nd International Conference on Extending Database Technology (EDBT 2019), 26–29 March 2019*, 2019.

[4] Angelos Charalambidis, Antonis Troumpoukis, and Stasinos Konstantopoulos. SemaGrow: Optimizing federated SPARQL queries. In *Proceedings of the 11th International Conference on Semantic Systems, SEMANTICS 2015, Vienna, Austria, September 15-17, 2015*, pages 121–128. ACM, 2015.

[5] Souripriya Das, Seema Sundara, and Richard Cyganiak. R2RML: RDB to RDF mapping language. W3C Recommendation 27 September 2012, September 2012.

[6] Olaf Görlitz and Steffen Staab. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *Proceedings of the 2nd International Workshop on Consuming Linked Data (COLD 2011), Bonn, Germany, October 23, 2011*, volume 782 of *CEUR Workshop Proceedings*, 2011.

[7] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 276–285, 1997.

[8] Desta Haileselassie Hagos, Theofilos Kakantousis, Vladimir Vlassov, Sina Sheikholeslami, Tianze Wang, Andrew Fleming, Andreas Cziferszky, Markus Muerth, Florian Appel, Antonis Troumpoukis, Stasinos Konstantopoulos, Despina-Athanasia Pantazi, Dimitris Bilidas, George Stamoulis, and Manolis Koubarakis. Platform software architecture – version 2. Technical Report Public Deliverable D1.6, Extreme Earth Project, December 2020.

[9] Olaf Hartig and Ralf Heese. The SPARQL query graph model for query optimization. In *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007, Proceedings*, pages 564–578, 2007.

[10] Stasinos Konstantopoulos, Angelos Charalambidis, Giannis Mouchakis, Antonis Troumpoukis, Jürgen Jakobitsch, and Vangelis Karkaletsis. Semantic Web technologies and Big Data infrastructures: SPARQL federated querying of heterogeneous Big Data stores. In *Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 19, 2016*, volume 1690 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[11] Stasinos Konstantopoulos, Angelos Charalambidis, Antonis Troumpoukis, Giannis Mouchakis, and Vangelis Karkaletsis. The Sevod vocabulary for dataset descriptions for federated querying. In *Proceedings of the 4th International Workshop on Dataset PROFIling and fEderated Search for Web Data (PROFILES 2017) co-located with The 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 22, 2017*, volume 1927 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

[12] Charalampos Kostopoulos, Giannis Mouchakis, Antonis Troumpoukis, Nefeli Prokopaki-Kostopoulou, Angelos Charalambidis, and Stasinos Konstantopoulos. KOBE: cloud-native open benchmarking engine for federated query processors. In *The Semantic Web - 18th International Conference, ESWC 2021, Virtual Event, June 6-10, 2021, Proceedings*, volume 12731 of *Lecture Notes in Computer Science*, pages 664–679. Springer, 2021.

[13] Kostis Kyzirakos, Manos Karpathiotakis, and Manolis Koubarakis. Strabon: A semantic geospatial DBMS. In *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, volume 7649 of *Lecture Notes in Computer Science*. Springer, 2012.

[14] Kostis Kyzirakos, Manos Karpathiotakis, and Manolis Koubarakis. Strabon: A semantic geospatial DBMS. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, volume 7649 of *Lecture Notes in Computer Science*, pages 295–311. Springer, 2012.

[15] Kostis Kyzirakos, Dimitrianos Savva, Ioannis Vlachopoulos, Alexandros Vasileiou, Nikolaos Karalis, Manolis Koubarakis, and Stefan Manegold. GeoTriples: Transforming geospatial data into RDF graphs using R2RML and RML mappings. *J. Web Semant.*, 52-53, 2018.

[16] Tanu Malik, Alexander S. Szalay, Tamas Budavari, and Ani Thakar. Skyquery: A web service approach to federate databases. In *First Biennial Conference on Innovative Data Systems Research, CIDR 2003, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*. www.cidrdb.org, 2003.

[17] Maroua Masmoudi, Mohamed Hedi Karray, Sana Ben Abdallah Ben Lamine, Bernard Archimede, and Hajer Baazaoui Zghal. Survey on semantic data integration approaches: Issues and directions. Articlle Under Review, Semantic Web Journal, 2021.

[18] Gabriela Montoya, Hala Skaf-Molli, and Katja Hose. The Odyssey approach for optimizing federated SPARQL queries. In *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, volume 10587 of *Lecture Notes in Computer Science*, pages 471–489. Springer, 2017.

[19] Open Geospatial Consortium. OGC GeoSPARQL: A geographic query language for RDF data, version 1.0. Open Geospatial Consortium Implementation Standard, OGC 11-052r4, September 2012.

[20] Claudia Paris, Lorenzo Bruzzone, Torbjørn Eltoft, Thomas Kræmer, Andrea Marinoni, Salman Khaleghian, Corneliu Octavian Dumitru, and Mihai Datcu. Large training database. Technical Report Public Deliverable D2.1, Extreme Earth Project, December 2019.

[21] Nefeli Prokopaki-Kostopoulou, Stasinos Konstantopoulos, Angelos Charalambidis, and Antonis Troumpoukis. Software for federating big linked geospatial data sources – version 1. Technical Report Public Deliverable D3.4, Extreme Earth Project, December 2019.

[22] Bastian Quilitz and Ulf Leser. Querying distributed RDF data sources with SPARQL. In *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*, volume 5021 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 2008.

[23] Paul Ramsey and Mark Leslie. Nearest-neighbour searching. Introduction to PostGIS, 2012. Online. Accessed: 2021-05-19.

[24] Muhammad Saleem and Axel-Cyrille Ngonga Ngomo. HiBISCuS: Hypergraph-based source selection for SPARQL endpoint federation. In *The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings*, volume 8465 of *Lecture Notes in Computer Science*, pages 176–191. Springer, 2014.

[25] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL query optimization. In Luc Segoufin, editor, *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*, ACM International Conference Proceeding Series, pages 4–33. ACM, 2010.

[26] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. FedX: Optimization techniques for federated query processing on linked data. In *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, pages 601–616. Springer, 2011.

[27] The PostGIS Development Group. ST_DWithin. PostGIS 3.1.2dev Manual, June 2021. Online. Accessed: 2021-06-15.

[28] Antonis Troumpoukis, Stasinos Konstantopoulos, Giannis Mouchakis, Nefeli Prokopaki-Kostopoulou, Claudia Paris, Lorenzo Bruzzone, Despina-Athanasia Pantazi, and Manolis Koubarakis. Geofedbench: A benchmark for federated GeoSPARQL query processors. In *Proceedings of the Posters and Demos Session of the 19th International Semantic Web Conference (ISWC 2020), 2-6 November 2020.*, 2020.

[29] Xin Wang, Thanassis Tiropanis, and Hugh C. Davis. LHD: optimising linked data query processing using parallelisation. In Christian Bizer, Tom Heath, Tim Berners-Lee, Michael Hausenblas, and Sören Auer, editors, *Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013*, volume 996 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[30] Chuanrong Zhang, Tian Zhao, Luc Anselin, Weidong Li, and Ke Chen. A map-reduce based parallel approach for improving query performance in a geospatial semantic web for disaster response. *Earth Sci. Informatics*, 8(3):499–509, 2015.

[31] Tian Zhao, Chuanrong Zhang, Luc Anselin, Weidong Li, and Ke Chen. A parallel approach for improving geo-sparql query performance. *Int. J. Digit. Earth*, 8(5):383–402, 2015.