CEUR-WS.org/Vol-3157/paper3.pdf

A Geospatial Join Optimization for Federated GeoSPARQL Querying

Antonis Troumpoukis[a], Stasinos Konstantopoulos[a] and Nefeli Prokopaki-Kostopoulou[b]

[a]*Institute of Informatics and Telecommunications, NCSR 'Demokritos', Ag. Paraskevi 15341, Greece*
[b]*Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Athens 16122, Greece*

Abstract

We present a join optimization method for federated linked geospatial data. This optimization targets *within-distance* queries in cases where the shapes that are compared are served from different sources of the federation. This operation is computationally expensive, because it cannot be answered from the spatial index of any of the individual data sources in the federation. Our optimization augments the subqueries prepared for each source with additional restrictions on topological relations that must be satisfied and that (a) can be answered from the spatial index of the individual data sources; (b) do not change the semantics of the query. We evaluate our optimization on data and queries derived from a real-world workflow over land usage data. Evaluation shows that our optimization substantially improves query processing time.

Keywords

Federated and distributed query processing, linked geospatial data, GeoSPARQL query processing

1. Introduction

Linked geospatial 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.

Indexing and *query optimization* are key features of all database management systems, both geospatial and relational. For instance, consider the query that retrieves *"all restaurants within 1km from Syntagma Square, Athens, Greece."* A centralized geospatial database relies on its geospatial index to exclude instances that cannot possibly be within 1km from Syntagma Square, so that the distance only needs to be computed for a small number of candidates. A query optimizer is aware that it is much more efficient to first retrieve the location of Syntagma Square

GeoLD 2022: 5th International Workshop on Geospatial Linked Data co-located with ESWC, May 30 2022, Hersonissos, Greece
antru@iit.demokritos.gr (A. Troumpoukis); konstant@iit.demokritos.gr (S. Konstantopoulos); nefelipk@di.uoa.gr (N. Prokopaki-Kostopoulou)
0000-0003-1078-8121 (A. Troumpoukis); 0000-0002-2586-1726 (S. Konstantopoulos); 0000-0002-8645-1341 (N. Prokopaki-Kostopoulou)
© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

and then filter for restaurants, than it would be to first retrieve the location of all restaurants in the world. So query optimization ensures that query execution is organized optimally and not following the ordering of the sub-queries in the original query.

Although both indexing and query optimization are based on mature technologies, it is not always straightforward to port these technologies to linked geospatial data and, even more so, to federated query processing of linked geospatial data. Suppose that we are federating a database that knows points of interest such as “Syntagma Square” with a database that knows commercial points of interest, including restaurants. There is no single geospatial index that can restrict the restaurants sub-query to the instances can potentially be solutions.

In the remainder of this paper, we present some background information (Section 2), and we describe our optimization for federated within-distance queries in detail (Section 3). We implement this technique in Semagrow and we compare the optimized version with the unoptimized one (Section 4). Finally, we discuss relevant work (Section 5) and conclude (Section 6).

2. Background

2.1. The GeoSPARQL query language

The GeoSPARQL specification [1] defines a set of classes and properties for asserting and querying geospatial information. Each feature is linked with a geometry with the `geo:hasGeometry` property[1], while the `geo:asWKT` property is used to provide the concrete geographical shape of a geometry as an RDF literal of the `geo:wktLiteral` datatype.[2] Given the above, the link between features and their concrete coordinates follows the pattern:

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

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 filter expressions that contain geospatial functions. For example:

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

uses the `geof:sfIntersects` function[3] 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. Notice that a geospatial join is essentially a cross product that is being filtered by a geospatial condition comparing values from both expressions.

2.2. Join evaluation in federated query processors

Bind join [2] is a join implementation suitable for distributed and federated environments because it is designed to drastically reduce the communication costs of joining two relations.

[1]We use `geo:` for the `http://www.opengis.net/ont/geosparql#` namespace.

[2]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 WTK serialization, and it is straightforward to transfer this discussion to GML or any other serialization.

[3]We use `geof:` for `http://www.opengis.net/def/function/geosparql/`

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 bind join [3, 4] 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 [5, 6]. Fortunately, SPARQL 1.1 specification foresees a VALUES keyword as a mechanism for passing multiple variable bindings at once. In addition, Schwarte et al. [6] proposed a technique that can simulate this behavior by 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.

Filter pushdown [7] 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 being 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.

3. Optimization of federated within-distance queries

3.1. Preliminaries

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, the *buffer* of size d around a is a spatial object that contains all points whose distances from a are 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 it 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.

3.2. Intuition and example

A *federated within-distance query* is a query that involves fetching pairs of shapes that are served from different federated sources and their distance is less or equal than a given length measurement. Since the GeoSPARQL specification does not offer any specialized function for the within-distance operator (contrary to e.g., PostGIS) 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, and the bindings of `?x` and `?y` are retrieved from different sources of the federation.

We design our optimization technique under the general assumption that the federation engine evaluates all federated joins in a bind join fashion with a filter pushdown optimization (cf. Subsection 2.2 for details). In this case, the federator operates by fetching "left-hand" WKTs to partially bind two-variable functions and then it pushes the filter to the "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.

Recall that in the naive implementation of bind join, for each binding of the left part of the join, the federator issues one query to the endpoint of the right part of the join. Assume, without loss of generality, that the bindings of `?x` are retrieved from the left part of the join. Thus, a query issued by the federator to the right-hand endpoint should contain a filter of the form:

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

where `WKT_LITERAL` is obtained by the bindings of the left part of the join. Such queries are usually slow, because the evaluation of this filter cannot use the spatial index of the source.

The evaluation of the source query in the remote endpoint can be sped-up if the federator adds an additional geospatial filter for filtering out all irrelevant candidate shapes for `?y`:

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

where `BBOX_BUF_LITERAL` is the minimum bounding box of the buffer of size d around `WKT_LITERAL`. This additional filter is evaluated by the geospatial source using its 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 the planning time.

In the following, we will extend this approach so as to apply in more elaborate implementations of bind join, where multiple bindings can be passed in a single query.

3.3. Join optimization algorithm

We assume the existence of some helper routines that their implementations are not included here as separate algorithms. Given a WKT literal a, a numeric literal d and a unit of measure u, the routine $\textsc{Buffer}(a, d, u)$ returns a WKT literal that represents the buffer of size d around a measured in units of type u. Given a WKT literal a, the routine $\textsc{Mbb}(a)$ returns a WKT literal that represents the minimum bounding box of a. Given an endpoint E, a query Q and a set of bindings B, the routine $\textsc{EvaluateRemoteQuery}(E, Q, B)$ implements the behavior of the query executor (cf. Subsection 2.2) and returns a set of bindings obtained from the source. Our

Algorithm 1 EvaluateRemoteQueryOpt

```
Input: a GeoSPARQL endpoint $E$, a GeoSPARQL query $Q$, and a set of bindings $B$.
Output: a set of bindings $R$
1: if $Q$ contains a filter of the form FILTER( geof:distance(?w1,?w2,$u$) < $d$ )
           where $u$ is a unit of measure URI, $d$ is a numeric literal,
           one of ?w1, ?w2 is a free variable w.r.t $B$ (denoted as $x$), and
           one of ?w1, ?w2 is a bound variable w.r.t $B$ (denoted as $y$) then
2:     $z :=$ a fresh variable
3:     Obtain $Q'$ by augmenting $Q$ with the filter FILTER( geof:sfIntersects($x$,$z$) )
4:     $B' := \{b \cup \{z/\textsc{Mbb}(\textsc{Buffer}(b(y), d, u))\} \,:\, b \in B\}$
5:     $R' := \textsc{EvaluateRemoteQuery}(E, Q', B')$ ,    $R := \{r - \{z/v\} \,:\, r \in R'\}$
6:     return $R$
7: else
8:     return $\textsc{EvaluateRemoteQuery}(E, Q, B)$
9: end if
```

goal is to extend EvaluateRemoteQuery with the optimization sketched in Subsection 3.2, adapted for the efficient bind join implementation that groups multiple query bindings from the left-hand endpoint in a single query for the right-hand endpoint. Thus, in Algorithm 1, we define EvaluateRemoteQueryOpt, which behaves as a wrapper for the original executor.

The first step (line 1) is to identify whether the input query is in the form of our interest. We consider *within-distance* joins (i.e., a filter of the form `geof:distance(?w1,?w2,`u`)` $<d$), and the join is a *federated* one (i.e., only one of the arguments `?w1`, `?w2` is bound from the bindings that came from the left part of the join). When the optimization is not applicable, EvaluateRemoteQueryOpt falls back to the original EvaluateRemoteQuery behavior.

The next step (lines 2-4) is to rewrite the query. After we identify that the free variable is the variable x and the bound variable is the variable y (line 1), we add an additional filter that forces x to intersect with the minimum bounding box of the buffer of size d around y (line 3). Notice though that unlike Subsection 3.2, we have several bindings for the variable y, thus we cannot hard-code the bounding box directly in the query. Instead, the additional filter is `geof:sfIntersects(`x,z`)`, and we provide additional bindings for the new fresh variable z (line 2) by extending each binding b from B with the 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 (line 4).

The final step (line 5) 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.

3.4. Correctness of the optimization

Lemma 1. *Let p, q be WKT literals and d be a length measurement. If the distance between p and q is less than d, then p intersects with the minimum bounding box of the buffer of size d around q.*

Proof. Let $p.q$ be geometric literals, such that the distance between p and q is less than d. Then, by definition, p intersects with the buffer of size d around q. Since all points of a shape s belong also to the minimum bounding box of s, we can conclude that p also intersects with the minimum bounding box of the buffer of size d around q. $\square$

Following Pérez et al. [8] we denote by $[\![\cdot]\!]_D$ the *evaluation* of a SPARQL query over a dataset D. If a query Q is a SELECT query, then $[\![Q]\!]_D$ is a set of bindings, which are the solutions that satisfy Q over D. Moreover, given a query Q and a query binding b, we write Qb to denote the query obtained by substituting each bound variable x of b in Q with $b(x)$. We assume set semantics of GeoSPARQL.

Lemma 2. *Let b be a binding, x be a free variable w.r.t. b, y be a bound variable w.r.t. b, u be a unit of measure URI, and d be a numeric literal. We define* $F_1 = (\texttt{geof:distance}(x, y, u) \ < d)$, $F_2 = \texttt{geof:sfIntersects}(x, z)$, *where* $z = \textsc{Mbb}(\textsc{Buffer}(b(y), d, u))$. *Let* Q_1, Q_2 *be two GeoSPARQL queries, s.t.* Q_2 *contains* F_1 *and* Q_1 *is obtained by augmenting* Q_2 *with* F_2. *Then,* $[\![Q_1 b]\!]_D = [\![Q_2 b]\!]_D$.

Proof. Suppose that the lemma does not hold. Since Q_1 is essentially Q_2 augmented with an additional filter, it must be $[\![Q_1 b]\!]_D \subseteq [\![Q_2 b]\!]_D$, i.e., there must exist some binding b^* such that $b^* \notin [\![Q_1 b]\!]_D$ and $b^* \in [\![Q_2 b]\!]_D$. According to the meaning of F_1, F_2, this means that (a) the distance between $b^*(x)$ and $b(y)$ is less than d and (b) $b^*(x)$ does not intersect with the minimum bounding box of a buffer of size d around $b(y)$. Contradiction (Lemma 1). $\square$

Theorem 1. *Let E be a GeoSPARQL endpoint, Q be a GeoSPARQL query, and B be a set of bindings. Then,* $\textsc{EvaluateRemoteQueryOpt}(E, Q, B) = \textsc{EvaluateRemoteQuery}(E, Q, B)$.

Proof. If Q is not of the form of the query in Line 1 of Algorithm 1, then the equation holds (cf. Line 8). Otherwise, notice that it holds $\mathscr{E}_{opt}(E, Qb, \emptyset) = \mathscr{E}(E, Qb, \emptyset)$ for all $b \in B$ (Lemma 2) (for brevity, we write $\mathscr{E}_{opt}$ and $\mathscr{E}$ instead of EvaluateRemoteQueryOpt and EvaluateRemoteQuery), respectively). Therefore, notice that for all $b \in B$ it holds $\mathscr{E}_{opt}(E, Q, \{b\}) = \mathscr{E}_{opt}(E, Qb, \emptyset) = \mathscr{E}(E, Qb, \emptyset) = \mathscr{E}(E, Q, \{b\})$, and thus $\mathscr{E}_{opt}(E, Q, B) = \bigcup_{b \in B} \mathscr{E}_{opt}(E, Q, \{b\}) = \bigcup_{b \in B} \mathscr{E}(E, Q, \{b\}) = \mathscr{E}(E, Q, B)$. $\square$

3.5. Implementation

We provide an implementation of our geospatial join optimization integrated in the Semagrow SPARQL federation engine.[4] Our implementation extends the existing query execution mechanism with a GeoSPARQL query executor that uses this optimization.

4. Evaluation

4.1. Experimental Setup

We evaluate our technique using datasets and queries from the GeoFedBench benchmark [9], which was derived from practical use cases in the agro-environmental domain; that is, linking land usage data with ground observations for the purpose of estimating crop type accuracy.

[4] Cf. https://github.com/semagrow/semagrow

Table 1
Dataset statistics.

dataset	# shapes	# all triples	# geospatial triples	# thematic triples	# properties
INVEKOS	2,008,137	14,056,959	4,016,274	10,040,685	7
LUCAS	4,325	30,379	8,650	21,729	11

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

1. The Austrian Land Parcel Identification System (INVEKOS)[5], 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)[6], which contains agro-environmental and soil data by field observation of geographically referenced points.

Table 1 gives more details about these datasets. Each dataset is loaded on a different SPARQL endpoint; we use the Strabon geospatial RDF store [10] for serving the data. Strabon encapsulates PostGIS for performing spatial operations, and uses a spatial index to optimize query time.

Federations We use two federations of the above datasets; one Semagrow federation in which the execution engine uses technique discussed in Section 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 2 we summarize the queries of the experiment. Each row of the table corresponds with a query template with a specific parameter. Q1-3 are used to estimate the crop-type reliability of the INVEKOS dataset. For each LUCAS instance URI, we check if it provides a positive validation (Q1), a negative validation (Q2), or it is irrelevant for the analysis (Q3). In order to get more information about the instances, each query either returns a single result (if the LUCAS URI has the desired property) or an empty result set (otherwise). These queries have several characteristics apart from federated within-distance geospatial joins (cf. [9] 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.

Experiment deployment and execution We use a Kubernetes 1.14 cluster with 1 master node and 8 worker nodes with a total of 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 [11], and the KOBE configurations for reproducing the experiments are publicly available.[7]

[5]cf. http://www.data.gv.at/katalog/dataset/f7691988-e57c-4ee9-bbd0-e361d3811641

[6]cf. https://esdac.jrc.ec.europa.eu/projects/lucas

[7]The experiment specifications can be found in https://github.com/semagrow/benchmark-geofedbench.

Table 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	-

4.2. Experimental Results

4.2.1. Data Validation Query Set (Q1-3)

In the first part of the experimental study, we compare the optimized (semagrow-opt) version of Semagrow over the unoptimized one (semagrow-std) using the query load obtained by the data validation task. Table 3 contains the experimental results. For every query template, we illustrate: the number of queries for each template (#queries), the number of queries that return result (#results), and the query processing time. For each federation, we display the total time to evaluate the query load and the average time for each query of the query load.

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 operation. This can be verified if we analyze the execution plan of each query. 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.

The previous discussion suggests that the operation “retrieve all WKTs from INVEKOS within 10m distance from a specific WKT (obtained from LUCAS)” is the costliest operation in the data

Table 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	396	82 hours	119 sec	99 mins	2.4 sec
Q3	2488	400	81 hours	117 sec	74 mins	1.8 sec

Table 4
Experimental results for Q4.

		semagrow-std		semagrow-opt			
query	distance	query proc. time	#results	query proc. time	shapes pruned by optimization		#results
Q4	10 m	58 sec	2	0.1 sec	2,008,134	(>99%)	2
Q4	100 m	57 sec	7	0.1 sec	2,008,129	(>99%)	7
Q4	1 km	58 sec	70	0.1 sec	2,008,004	(>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

validation task. As a result, it is safe to conclude that our optimization technique which targets such queries is the reason for the extreme speed-up of the INVEKOS validation task.

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

In the second part of the experimental study, we compare the optimized (semagrow-opt) over the unoptimized (semagrow-std) version of Semagrow using a query of fetching all INVEKOS parcels that are found within increasing distances from a specific LUCAS point. Table 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 to the right part of the federated join.

First, we notice that in both cases, Semagrow returns the same number of results. This validates that the implementation of the optimization is correct, i.e., the bounding box in the additional filter is constructed correctly, thus the filter does not prune any correct shapes.

Regarding semagrow-std, which issues the unoptimized query in INVEKOS, we notice that every query requires at least 57 seconds. For a parameterized distance between 10 meters and 10 kilometers, the query time is 57 – 58 seconds, even if the result is either very small (2 results) or moderate (4,700 results). Only for distances greater than 50 kilometers, where we have a very large result set (greater than 140,000 results), the query time seems to increase as we increase the distance parameter. This behavior can be explained if we consider that the unoptimized query cannot be answered using the spatial index of the source, and the source has to check for every shape in INVEKOS if it is within a specific distance. Thus, the experimental results can be explained as follows; the time needed to check potential candidates within a given distance is around 1 minute, and the remaining time is used for passing the results back to the clients. Notice that in our case, the transfer cost can be relatively high (e.g., 50 seconds for 500,000 results), because the result set includes WKT values which are in general long strings.

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 parameter, i.e., for smaller distances we have a faster query processing time. Recall that the optimized source query has an additional filter expression, which is used to prune all shapes that are too far away from the given LUCAS WKT. Indeed, we observe that there exists a connection between the amount of the pruning of the additional filter and the query processing time. Unlike previously, where the source had to consider all shapes from INVEKOS, the source here computes the distance for a reduced set of candidate shapes, which is relevant to the distance parameter. 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.

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 [12, Section 5] find that there is no mature federated GeoSPARQL query processing system. Recent work on data integration methods cites systems that collect and integrate distributed geospatial data into a single store as well dynamic federation of non-geospatial data sources, but also does not include systems that are both federated and support geospatial operations [13, Sect. 2].

PostGIS already has a build-in function calculating the geometries that are within a given distance, called `ST_DWithin` [14], 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 [15]. 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 an 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 [16]. 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 [16] and parallelizing them [17]. 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.

6. Conclusions and Future Work

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, without changing the semantics of the original query. 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 substantially speeds up query execution, especially for restrictive within-distance limits. This can prove useful for many real-world applications, as it is reasonable to expect that within-distance restrictions are used to limit results to a local scope. In other words, the value of the proposed method might be smaller for within-distance limits of thousands of kilometers, but it is hard to imagine use cases where such queries would be useful.

As future work, we plan to develop a GeoSPARQL extension where within-distance restrictions are expressed with a single function. This will allow the scope of application of our optimization to be made explicit, and not depend on statically analysing the query in order to identify queries where the optimization is applicable. Another direction for future work is to develop similar rewriting techniques for optimizing federated queries with other GeoSPARQL functions (such as `geof:sfCrosses` or `geof:sfTouches`). Finally, an interesting open question is whether (and to what extend) our approach can be effective for triple stores that do not provide a complete implementation of the GeoSPARQL standard.

Acknowledgements The work described here has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825258. For more details, please visit http://earthanalytics.eu. The authors also acknowledge that the work was supported by SKEL, NCSR 'Demokritos' https://www.iit.demokritos.gr/labs/skel/.

References

[1] Open Geospatial Consortium, OGC GeoSPARQL: A geographic query language for RDF data, version 1.0, 2012. URL: http://www.opengis.net/doc/IS/geosparql/1.0.

[2] L. M. Haas, D. Kossmann, E. L. Wimmers, J. Yang, Optimizing queries across diverse data sources, in: Proceedings of VLDB'97, 1997. URL: http://www.vldb.org/conf/1997/P276.PDF.

[3] O. Görlitz, S. Staab, SPLENDID: SPARQL endpoint federation exploiting VOID descriptions, in: Proceedings of the 2nd Intl. Workshop on Consuming Linked Data COLD, 2011. URL: http://ceur-ws.org/Vol-782/GoerlitzAndStaab_COLD2011.pdf.

[4] B. Quilitz, U. Leser, Querying distributed RDF data sources with SPARQL, in: Proc. of ESWC 2008, Tenerife, Canary Islands, Spain, 2008. doi:10.1007/978-3-540-68234-9_39.

[5] A. Charalambidis, A. Troumpoukis, S. Konstantopoulos, SemaGrow: Optimizing federated SPARQL queries, in: Proc. of SEMANTICS 2015, 2015. doi:10.1145/2814864.2814886.

[6] A. Schwarte, P. Haase, K. Hose, R. Schenkel, M. Schmidt, FedX: A federation layer for distributed query processing on linked open data, in: Proceedings of ESWC 2011, Heraklion, Crete, Greece, 2011. doi:10.1007/978-3-642-21064-8_39.

[7] M. Schmidt, M. Meier, G. Lausen, Foundations of SPARQL query optimization, in: Proceedings of ICDT 2010, Lausanne, Switzerland, 2010. doi:10.1145/1804669.1804675.

[8] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, ACM Trans. Database Syst. (2009). doi:10.1145/1567274.1567278.

[9] A. Troumpoukis, S. Konstantopoulos, G. Mouchakis, N. Prokopaki-Kostopoulou, C. Paris, L. Bruzzone, D.-A. Pantazi, M. Koubarakis, GeoFedBench: A benchmark for federated GeoSPARQL query processors, in: Posters and Demos of ISWC 2020, Virtual Event, 2020.

[10] K. Kyzirakos, M. Karpathiotakis, M. Koubarakis, Strabon: A Semantic Geospatial DBMS, in: Proc. of ISWC 2012, Boston, MA, USA, 2012. doi:10.1007/978-3-642-35176-1_19.

[11] C. Kostopoulos, G. Mouchakis, A. Troumpoukis, N. Prokopaki-Kostopoulou, A. Charalambidis, S. Konstantopoulos, KOBE: Cloud-native open benchmarking engine for federated query processors, in: Proc. of ESWC 2021, 2021. doi:10.1007/978-3-030-77385-4_40.

[12] K. Bereta, H. Caumont, U. Daniels, E. Goor, M. Koubarakis, D. Pantazi, G. Stamoulis, S. Ubels, V. Venus, F. Wahyudi, The Copernicus App Lab project: Easy access to Copernicus data, in: Proceedings of EDBT 2019, Lisbon, Portugal, 2019. doi:10.5441/002/edbt.2019.46.

[13] M. Masmoudi, S. B. A. B. Lamine, H. B. Zghal, B. Archimède, M. Karray, Knowledge hypergraph-based approach for data integration and querying: Application to Earth Observation, Future Gener. Comput. Syst. (2021). doi:10.1016/j.future.2020.09.029.

[14] PostGIS Manual, ST_DWithin, 2021. URL: https://postgis.net/docs/ST_DWithin.html.

[15] P. Ramsey, M. Leslie, Nearest-neighbour searching, PostGIS, 2012. URL: https://www.postgis.net/workshops/postgis-intro/knn.html, accessed: 2021-05-19.

[16] T. Zhao, C. Zhang, L. Anselin, W. Li, K. Chen, A parallel approach for improving geosparql query performance, Int. J. Digit. Earth (2015). doi:10.1080/17538947.2014.904012.

[17] C. Zhang, T. Zhao, L. Anselin, W. Li, K. Chen, A map-reduce based parallel approach for improving query performance in a geospatial semantic web for disaster response, Earth Sci. Informatics (2015). doi:10.1007/s12145-014-0179-x.