A Geospatial Join Optimization for Federated GeoSPARQL querying

Antonis Troumpoukis, Stasinos Konstantopoulos, and
Nefeli Prokopaki-Kostopoulou

Institute of Informatics and Telecommunications, NCSR ”Demokritos”

5th International Workshop on Geospatial Linked Data (GeoLD2022)
May 30, 2022, Heraklion, Crete, Greece

Outline

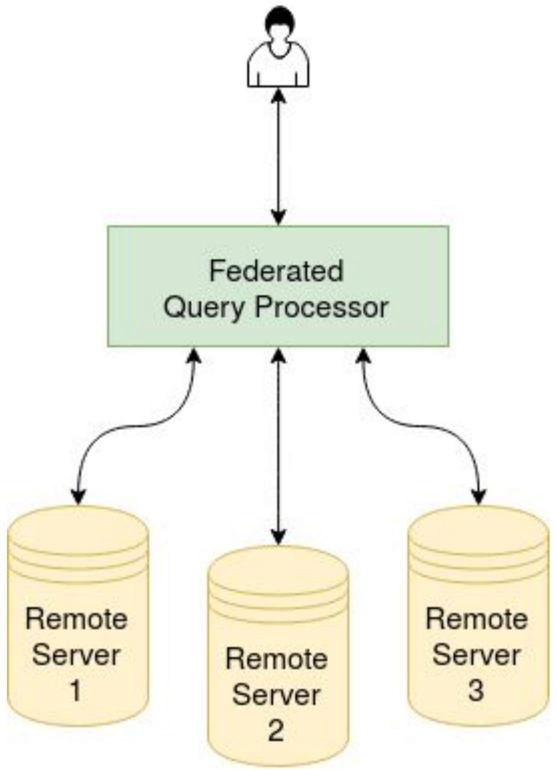
- Background: Federated query processing
- Optimization of federated within-distance queries
- Evaluation (using a real-world use case)
- Conclusions and future work

Outline

- **Background: Federated query processing**
- Optimization of federated within-distance queries
- Evaluation (using a real-world use case)
- Conclusions and future work

Federated query processing

- *Federated query processors* are systems that seamlessly integrate data from multiple remote dataset servers
- receive a query, issue the necessary subqueries in the remote servers, combine the intermediate results accordingly, and presents the result to the client.
- used thoroughly in Linked Data; there exist many data providers that publish their (thematic) datasets through public SPARQL endpoints
- the technology is not yet mature for Geospatial Linked Data and GeoSPARQL endpoints



Federated Geospatial Joins

- GeoSPARQL specification:
  - geospatial operations are denoted using functions between geographic literals (e.g., **geof:sfWithin**, **geof:sfIntersects, geof:distance**)
  - geographic literals are denoted using WKT serializations (e.g., “POINT(21.814 38.422)”^^**geo:wktLiteral**)
  - features are linked with corresponding geographic literals using **geo:hasGeometry**/**geo:asWKT** chains

```
SELECT * WHERE
{
   ?s1 geo:hasGeometry ?g1 .
   ?g1 geo:asWKT ?w1 .
   ?s2 geo:hasGeometry ?g2 .
   ?g2 geo:asWKT ?w2 .
   FILTER ( geof:sfIntersects(?w1, ?w2) )
}
```

- A geospatial join is a *cross product* filtered by a *geospatial function*.
- A federated geospatial join is a *cross product* filtered by a *geospatial function* comparing shapes coming from different endpoints.

Federated Join Implementations

Federated Thematic Joins

- Many algorithms and implementations exist (*bind join*, *hash join*, *adaptive join*, etc.)
- Bind Join!
  - issue a query to the “left” endpoint, then pass its results as bindings to the “right” endpoint.
  - reduces the communication cost by reducing intermediate results.

Federated Geospatial Joins

- No specialized algorithms for federated geospatial joins exist
- Bind Join with a FILTER pushdown!
  - fetch “left” shapes to partially bind the geospatial function, then push the filter to the “right” endpoint
  - exploits the fact that geospatial functions are evaluated faster in the sources (spatial index).

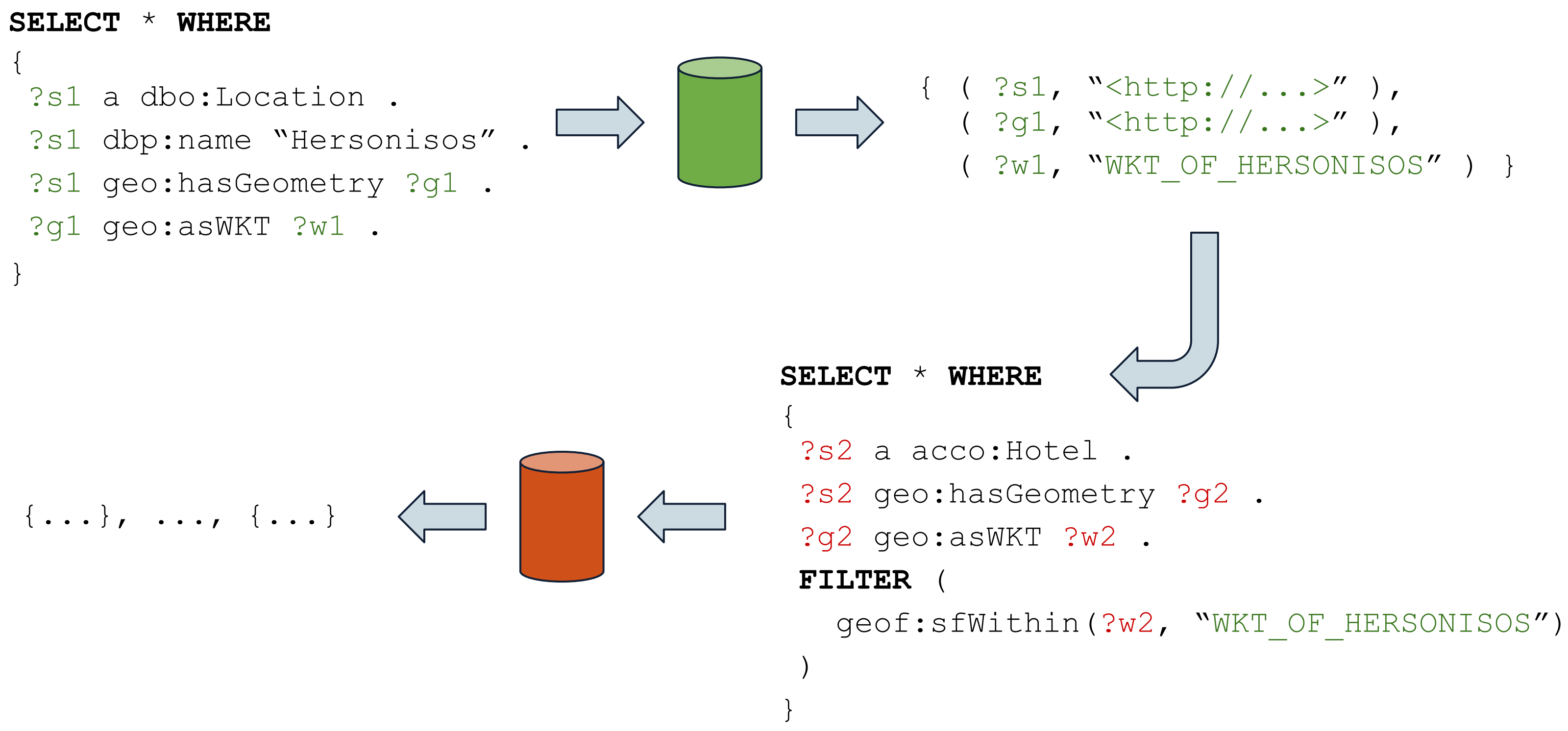
Example

"Given 2 endpoints (administrative divisions, hotels), fetch all hotels **within** Hersonisos"

```
SELECT * WHERE
{
 # Hersonisos and its geometry
 ?s1 a dbo:Location .
 ?s1 dbp:name "Hersonisos" .
 ?s1 geo:hasGeometry ?g1 .
 ?g1 geo:asWKT ?w1 .

 # Hotels and their geometries
 ?s2 a acco:Hotel .
 ?s2 geo:hasGeometry ?g2 .
 ?g2 geo:asWKT ?w2 .

 # hotel is within Hersonisos
 FILTER ( geof:sfWithin(?w2, ?w1) )
}
```



Outline

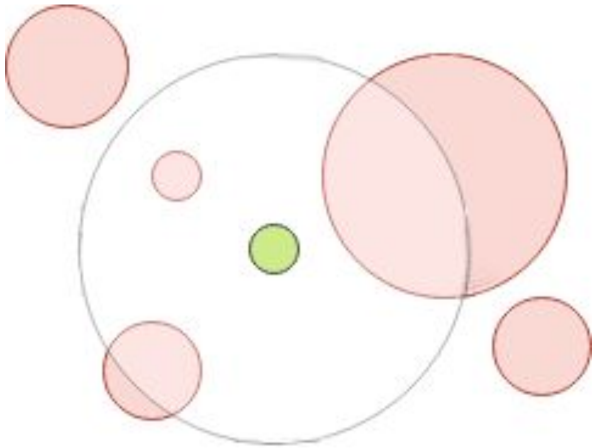
- Background: Federated query processing
- **Optimization of federated within-distance queries**
- Evaluation (using a real-world use case)
- Conclusions and future work

Federated Within-Distance Queries

- We reduce focus on **federated within-distance** queries:
  - shapes from different endpoints that their distance is less than d
  - without requiring the exact distance

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

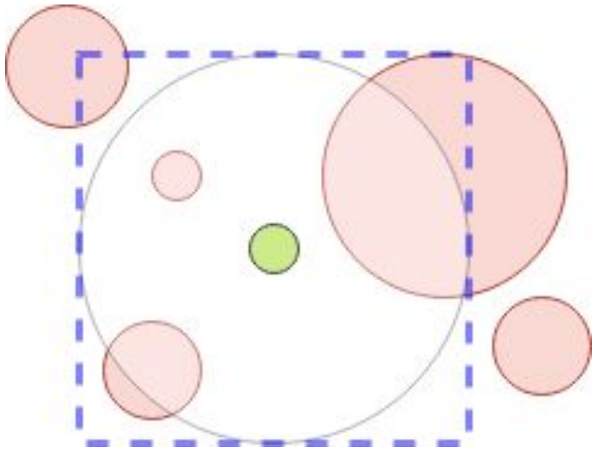
- **Problem:** The evaluation such filters is computationally expensive:
  - it cannot be answered from the spatial index.
  - every shape is a potential match and its distance should be compared with the threshold.



Example: The process of finding all red shapes within distance d from the given green shape, is slow. For each shape we have to calculate the distance from the given shape and compare it with the threshold d.

Optimizing Federated Within-Distance Queries

- **Solution**: We augment the subquery to be issued to the "right-hand" endpoint with an additional FILTER:
  - keeps only shapes that do not intersect with a constructed rectangle
  - used to prune all shapes that are "too-far away"
  - can be answered from the spatial index of the source.
- We efficiently refine the set of candidate shapes before starting to actually compute distances.



Example: To speed up the process of finding all red shapes within distance d from the given green shape, we insert a condition that filters out all shapes that do not intersect with the blue rectangle.

Example

"Given 2 endpoints (administrative divisions, hotels), fetch all hotels **within 1km distance** from Hersonisos"

```
SELECT * WHERE
{
 # Hersonisos and its geometry
 ?s1 a dbo:Location .
 ?s1 dbp:name "Hersonisos" .
 ?s1 geo:hasGeometry ?g1 .
 ?g1 geo:asWKT ?w1 .

 # Hotels and their geometries
 ?s2 a acco:Hotel .
 ?s2 geo:hasGeometry ?g2 .
 ?g2 geo:asWKT ?w2 .

 # within 1 km distance
 FILTER ( geof:distance(?w1, ?w2,
   uom:metre) < 1000 ).
}
```

```
SELECT * WHERE
{
 ?s1 a dbo:Location .
 ?s1 dbp:name "Hersonisos" .
 ?s1 geo:hasGeometry ?g1 .
 ?g1 geo:asWKT ?w1 .
}
```

```
{ ( ?s1, "<http://.../>" ),
  ( ?g1, "<http://.../>" ),
  ( ?w1, "WKT_OF_HERSONISOS" ) }
```

```
BUFBBOX = MinBBox(Buffer("WKT_OF_HERSONISOS",
                  1000, uom:metre))
```

```
SELECT * WHERE
{
 ?s2 a acco:Hotel .
 ?s2 geo:hasGeometry ?g2 .
 ?g2 geo:asWKT ?w2 .
 FILTER ( geof:sfIntersects(?w2, "BUFBBOX") )
 FILTER (
   geof:distance("WKT_OF_HERSONISOS",
     ?w2, uom:metre) < 1000 ).
}
```

```
{...}, ..., {...}
```

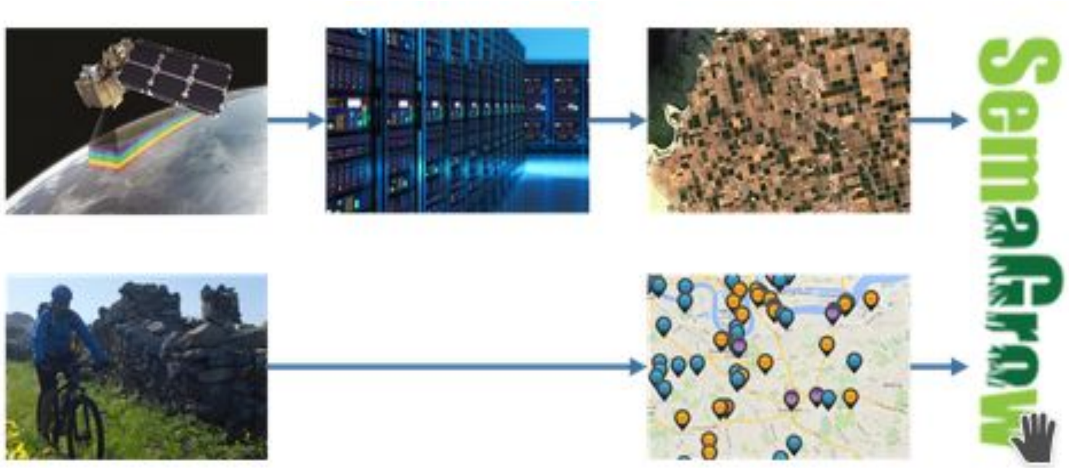
Contributions

- Optimization technique
  - We provide a pseudocode of our approach in the paper
  - The actual algorithm is slightly more complex (designed to work with multiple bindings per query).
- Correctness proof
  - We show that the additional FILTER does not change the semantics of the original query (it does not prune any unwanted shapes)
- Implementation
  - We provide an open source implementation of the technique
  - The implementation is integrated within the *Semagrow* federation engine

Outline

- Background: Federated query processing
- Optimization of federated within-distance queries
- **Evaluation (using a real-world use case)**
- Conclusions and future work

Experimental Evaluation
Validating Crop type data using Ground Observations



- A federation with 2 GeoSPARQL endpoints:
  - INVEKOS (field parcels, owners' self declaration)
  - LUCAS (Ground observations of crop type data)
- 14.1 million triples, ~4GB of data in N-triples format
- Task: Estimate the **crop-type accuracy** of the INVEKOS

Queries		
Q1	given a ground observation, return the *closest* field if it is within 10 meters and the crop types **match**	**positive**
Q2	given a ground observation return the *closest* field if it is within 10 meters and their crop types **do not match**	**negative**
Q3	given a ground observation, return it if there is *no* field within 10 meters	**irrelevant**

Experimental Results
Crop-type data validation task

	#queries in the workload	naive	optimized
		query exec. time (average per query)	query exec. time (average per query)
Q1	2488	120 sec	***2.6 sec***
Q2	2488	119 sec	***2.4 sec***
Q3	2488	117 sec	***1.8 sec***
		query exec. time (total workload)	**query exec. time (total workload)**
Q1-3	7494	10 days & 6 hours	***4 hours & 39 min***

- We evaluate the full workload for the data validation task.
- **optimized** is faster than **naive** by *2 orders of magnitude*.
- The queryset has several complex characteristics, but the bottleneck is the within-distance operation.
- Possible reason: the distance parameter is quite small (10 meters).

Experimental Results (cont.)
Additional query

	distance	naive	optimized	
		query exec. time	query exec. time	shapes pruned
Q4	10 m	58 sec	***0.1 sec***	>99%
Q4	100 m	57 sec	***0.1 sec***	>99%
Q4	1 km	58 sec	***0.1 sec***	>99%
Q4	10 km	57 sec	***1.2 sec***	99%
Q4	50 km	72 sec	***26 sec***	84%
Q4	100 km	110 sec	***86 sec***	60%

Q4	return all fields within-distance D from a specific ground observation (D = 10m-100km)

- **naive**: ~1 min to calculate the result, remaining time to fetch the result.
- **optimized**: smaller distance parameter → higher amount of pruning → faster query execution time. The additional filter does not introduce any time overheads.
- huge time difference for distance ≤1km, less pronounced for distance ≥50km.

Outline

- Background: Federated query processing
- Optimization of federated within-distance queries
- Evaluation (using a real-world use case)
- **Conclusions and future work**

Conclusions

- We proposed an optimization for federated geospatial within-distance joins
  - Augments the subqueries prepared for each source with additional filters that can be answered from the spatial index of the federated sources.
  - Does not change the semantics of the query
  - Implemented within the Semagrow federation engine.

- We show in our evaluation that the optimization substantially speeds-up query execution
  - we used datasets and queries from a practical use-case from the agro-environmental domain
  - very effective, especially for small distance limits (2 orders of magnitude for 1km)
  - can be useful for real-world applications (restrictions used to limit results to a local scope)

Future Work

- Develop a GeoSPARQL extension where within-distance is expressed with a single function.

- Develop similar rewriting techniques for optimizing queries with other GeoSPARQL functions.

Thank you!

Visit us at: https://github.com/semagrow/

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/.