Scalable Spatial Join for WFS Clients

Tian Zhao
University of Wisconsin – Milwaukee, Milwaukee, WI, USA
tzhao@uwm.edu

Chuanrong Zhang
University of Connecticut, Storrs, CT, USA
chuanrong.zhang@uconn.edu

Zhijie Zhang
University of Connecticut, Storrs, CT, USA
zhijie.zhang@uconn.edu

Abstract

Web Feature Service (WFS) is a popular Web service for geospatial data, which is represented as sets of features that can be queried using the *GetFeature* request protocol. However, queries involving spatial joins are not efficiently supported by WFS server implementations such as GeoServer. Performing spatial join at client side is unfortunately expensive and not scalable. In this paper, we propose a simple and yet scalable strategy for performing spatial joins at client side after querying WFS data. Our approach is based on the fact that Web clients of WFS data are often used for query-based visual exploration. In visual exploration, the queried spatial objects can be filtered for a particular zoom level and spatial extent and be simplified before spatial join and still serve their purpose. This way, we can drastically reduce the number of spatial objects retrieved from WFS servers and reduce the computation cost of spatial join, so that even a simple plane-sweep algorithm can yield acceptable performance for interactive applications.

2012 ACM Subject Classification Information systems → Geographic information systems

Keywords and phrases WFS, SPARQL, Spatial Join

Digital Object Identifier 10.4230/LIPIcs.GIScience.2018.72

Category Short Paper

1 Introduction

OGC Web services such as Web Feature Service (WFS) and Web Map Service (WMS) provide standard Web-based protocols for querying geospatial features. WMS clients can use *GetMap* request to retrieve map images for a specified area and use *GetFeatureInfo* request to query the attributes of specified features. WFS clients can use *GetFeature* request to retrieve the feature instances including the geometries and other feature attributes. The retrieved features can be used by clients for computations such as spatial joins.

In query-based visual exploration, users rely on an interactive client application to locate data of interests, where spatial join is a commonly used operation to discover spatial relations between features on a map. Spatial join is a computationally intensive operation that is usually executed in a server such as PostGIS database. Previous studies have focused on improving response time at server side [6] while very few research is on improving performance at client side [5]. However, in some cases, it is preferable to perform spatial joins at client side. For example, to join two or more types of features located in different WFS servers, it is inefficient to retrieve one set of features from one server and send them to the second server for spatial join. Moreover, WFS servers may not even provide efficient implementation of

Table 1 Selected features of high-definition hydrology dataset for HUC0204 – Delaware-Mid Atlantic Coastal sub-region in the Mid-Atlantic Water Resource Region.

Feature Type	Geometry Type	Number of Features	Shapefile Size
NHDFlowline	MultiLineString	310835	312 MB
NHDWaterbody	MultiPolygon	57641	151 MB
NHDLine	MultiLineString	8090	3.7 MB
NHDArea	MultiPolygon	2592	113 MB
NHDPoint	Point	514	0.023 MB

spatial joins. For example, GeoServer implements spatial joins of two layers as a *GetFeature* request to the first layer where the join operation with the second layer is encoded in the filter of the request. This is similar to the *nested-loop* join [4], which loads all features in the server memory and performs spatial join on each pair of features in the two layers. This is inefficient. For example, to avoid using too much server memory, GeoServer restricts the number of features in the filter to be 1000 or less by default, which limits its ability of handling big spatial datasets.

Implementing efficient spatial join at WFS clients requires special care. WFS GetFeature requests can overwhelm both the server and the client when involving in big spatial datasets. For example, the hydrology dataset shown in Table 1 contains features (e.g. flowline) over 300 megabytes (MB). If we make a request to retrieve all feature instances of the flowline layer, then either the WFS server will fail to respond or the browser that runs the WFS client will quit working due to memory exhaustion. Note that while WMS can build and return maps of a large number of features such as the aforementioned flowline, WFS needs to encode all feature data in a *GetFeature* response, which can severely strain the memory capacity of the server. The WFS client, which often is the Web or mobile browser, will also become overwhelmed by the amount of memory and computation workload that are required to decode the response, store the spatial objects, and display them on a map. In addition, transmitting hundreds of MB of data across network consumes time and bandwidth. Finally, even if the server and client can process the *GetFeature* requests without crashing, spatial join can still take exceedingly long time, which is unsuitable for an interactive application.

While it is possible to improve the runtime of spatial join by implementing more efficient spatial join algorithms, there is a limit on how much improvement one can make. The query response time for a WFS *GetFeature* request includes the query processing time at WFS server, network transmission time of the query response, decoding time of the response, and computation time of spatial join. Improving performance on spatial join alone will not be sufficient if the network time and server time are still significant. In addition, WFS clients are often implemented as dynamic Web pages running in browsers, where the join operation is implemented in JavaScript that runs as a single-threaded program. There is a limited opportunity to improve spatial join performance through parallelization.

Although many spatial join algorithms have been proposed in literature, very few studies investigated performance of spatial join query on the client side over the Web or mobile browser. Based on our best knowledge, there is no study to compare performance of different spatial join algorithms for WFS, not to say in the context of Geospatial Semantic Web. To address the above efficiency problem with spatial join at WFS clients, we propose an approach that leverages the fact that users of the WFS clients are mostly interested in features of the current map extent and zoom level by not retrieving irrelevant features before performing spatial joins. In addition, retrieved features are cached to reduce network traffic

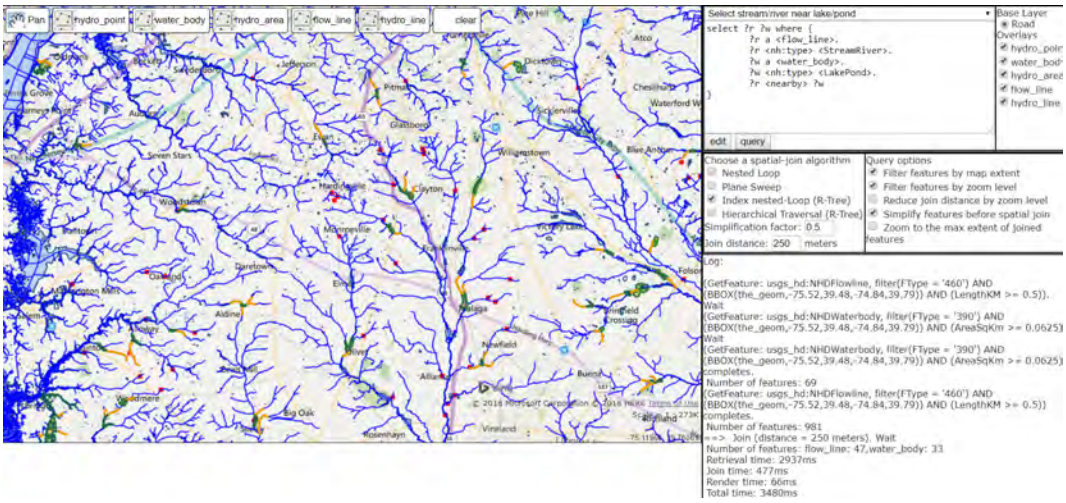


Figure 1 WFS query interface.

and server load and geometry simplification is used to improve the efficiency of evaluating spatial relations. Alternative spatial join algorithms are evaluated. For index-based join, spatial indices generated online are cached to reduce runtime costs.

For the rest of the paper, we first explain our approach in Section 2, then we evaluate its performance in Section 3, and we discuss the result in Section 4.

2 Approach

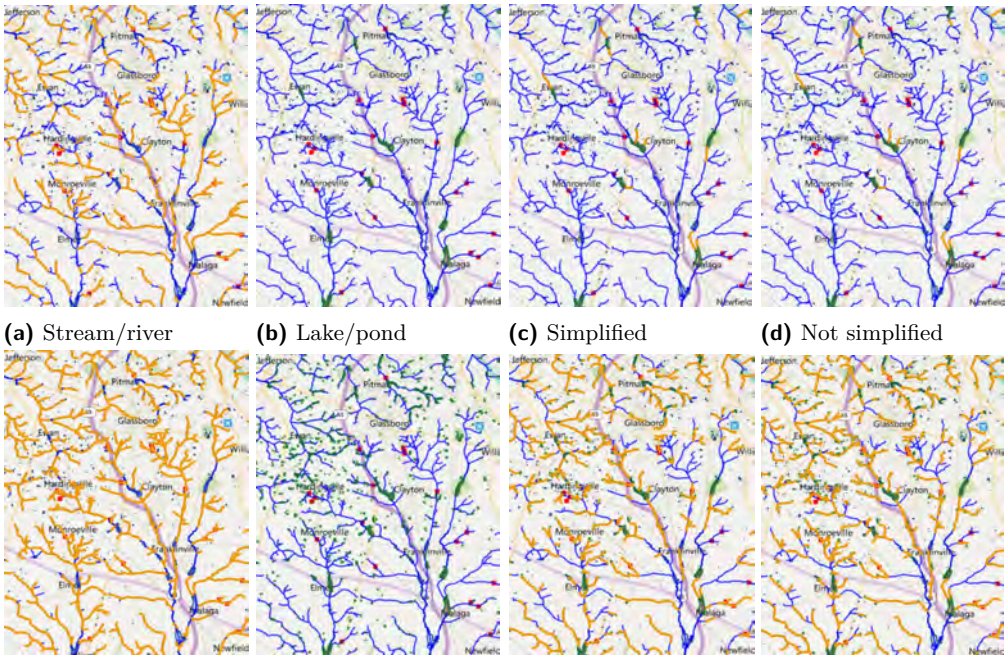
This WFS query client is an extension of our prior work on RDF query interface for WFS data [8, 7]. The spatial query is written in SPARQL-like syntax, which is translated to WFS requests and spatial join operations. A configuration file is used to map the WFS feature types and attributes to RDF classes and properties. Furthermore, certain attribute values are mapped to more recognizable constants for the convenience of writing queries. The query interface is shown in Figure 1 and the application is available at `http://tianpar.cs.uwm.edu:8080/usgs`.

The interface has an option to automatically insert spatial filters based on the current map extent so that features beyond the current map extent will not be retrieved. It also has an option to insert spatial filters based on the current zoom level so that features with sizes that are smaller than a threshold will not be retrieved. The threshold is calculated based on an adjustable scale proportional to the current zoom level. In order to perform spatial filtering based on feature size, we encode the attribute information of a feature type used to represent sizes in the configuration file.

For example, the query (Q1) below retrieves streams/rivers near lakes/ponds.

```
select ?r ?w where {                                          (Q1)
    ?r a <flow_line>.
    ?r <nh:type> <StreamRiver>.
    ?w a <water_body>.
    ?w <nh:type> <LakePond>.
    ?r <nearby> ?w            }
```

In this query, the predicate `<nearby>` specifies a spatial join between the variables `?r` and `?w`, which refer to features of streams/rivers and lakes/ponds respectively. The adjustable distance of `<nearby>` is specified separately in the query interface. This query is translated to the following concrete actions.



(a) Stream/river **(b)** Lake/pond **(c)** Simplified **(d)** Not simplified

(e) Stream/river† **(f)** Lake/pond† **(g)** Simplified† **(h)** Not simplified†

Figure 2 Part of selected streams/rivers (in yellow) and lakes/ponds (in green) with or without size filters (marked with †) and the join result with or without simplification.

```
(GetFeature: usgs_hd:NHDFlowline,
    filter(FType = '460') AND (BBOX(the_geom,-75.52,39.48,-74.84,39.79))
                          AND (LengthKM >= 0.5)).
(GetFeature: usgs_hd:NHDWaterbody,
    filter(FType = '390') AND (BBOX(the_geom,-75.52,39.48,-74.84,39.79))
                          AND (AreaSqKm >= 0.0625)).
Join (distance = 250 meters).
```

The extent and size filters are inserted automatically by the query client into the generated GetFeature requests. The extent filter `BBOX(the_geom,~-75.52,39.48,-74.84,39.79))` is derived from the current map extent (as in Figure 1). The size filter is derived from the current zoom-level and related to size attribute of each feature type. For streams/rivers, size filter is LengthKM ≥ 0.5 and for lakes/ponds, the size filter is AreaSqKm ≥ 0.0625.

We have implemented four spatial join algorithms: nested-loop join [4], plane sweep [1], index nested-loop join [3], and hierarchical traversal [2], where the latter two use R-tree indexing. Before spatial join, complex geometries (multi-line-strings and multi-polygons) are simplified based on a tolerance value proportional to the join distance. The implementation of nested-loop join is optimized by filtering candidate pairs using their bounding boxes.

3 Evaluation

We evaluated the performance of the four spatial join algorithms implemented in JavaScript running in Chrome browser. We used the query (Q1) to select the streams/rivers (NHDFlowline) that are within 250 meters of lakes/ponds (NHDWaterbody). Figure 2 shows some of the streams/rivers and lakes/ponds with or without size filters and the corresponding join results. The right two maps of each row are the join results with or without simplification. The maps on the second row (without size filters) are cluttered with features (especially

Table 2 Numbers of features after join, written as (# of streams/rivers, # of lakes/ponds).

Algorithm	Nested loop	Plane sweep	Index nested-loop	Hierarchical traversal
Filter by extent	(3192, 1764)	(3192, 1764)	(3192, 1764)	(3192, 1764)
Filter by extent & Simplify	(2865, 1514)	(2911, 1649)	(2865, 1514)	(2865, 1514)
Filter by extent/size	(51, 36)	(51, 36)	(51, 36)	(51, 36)
Filter by extent/size & Simplify	(47, 33)	(46, 33)	(47, 33)	(47, 33)

Table 3 Number of retrieved features and runtime (in seconds) for data retrieval, rendering results, and computing geometry bounds (included in the runtime of spatial join).

	Stream/River	Lake/Pond	Retrieval	Render	Bounds
Filter by extent only	8291	2757	11.6 s	0.85 s	0.99 s
Filter by extent and size	981	69	2.9 s	0.07 s	0.32 s

lakes/ponds) that are too small for visual exploration. From the figure it can be seen that, the join results with or without simplification ((c) vs. (d) and (e) vs. (f)) do not show obvious visual differences.

To measure the accuracy of various query options, we report in Table 2 the number of joined features that are with or without size filters and with or without simplification (with the tolerance of 125 meters). From Table 2, it can be seen that all four algorithms report similar results. The only exception is the *plane sweep* algorithm when the feature geometries are simplified. This difference is due to the combined effect of the simplification and the order of comparison of the join algorithms. Without simplification, all four algorithms report the same results. Simplification also reduces the number of joined features moderately.

Table 3 shows the number of retrieved features with or without size filters and the corresponding runtime for data retrieval, rendering results, and calculating geometry bounds. Table 4 shows the runtime of the four join algorithms for query (Q1) that are with or without size filters and with or without simplification. The runtime includes one-time costs such as calculating geometry bounds, simplification, spatial indexing (for index nested-loop join and hierarchical traversal), and sorting (for plane sweep). The costs are one-time since the bounds or indices are stored with the cached features and if the next user query uses cached data, such costs will not be repeated. Since these one-time costs are significant portions of the join time, for queries that can find data in the cache, the join time is much lower.

Caching reduces runtime cost even for queries that share some of the data. For example, if we first run the below query (Q2) and then run (Q1) with the same extent and size filters, the execution of (Q1) will be much faster because the features of streams/rivers will be in cache where spatial indices and geometry bounds have already been computed.

```
select ?r ?p where {                                                    (Q2)
        ?r a <flow_line>.
        ?r <nh:type> <StreamRiver>.
        ?p a <hydro_point>.
        ?r <nearby> ?p              }
```

In this case, the query (Q1) only needs to send a WFS request to retrieve lakes/ponds features and to perform spatial join. The runtime cost of (Q1) (with simplification) reduces from 3.5 s to about 1.7 s (1.5 s for data retrieval, 0.16 s for index nested-loop join – 0.12 s of which is for computing geometry bounds of lakes/ponds, while rendering is still 0.07 s).

Table 4 Runtime (in seconds) of spatial join (including one-time costs such as calculating geometry bounds, simplification, indexing, and sorting).

Algorithm	Nested loop	Plane sweep	Index nested-loop	Hierarchical traversal
Filter by extent	73.2 s	2.87 s	2.5 s	2.74 s
Filter by extent & Simplify	49.7 s	1.4 s	1.4 s	1.37 s
Filter by extent/size	3.9 s	0.76 s	0.75 s	0.72 s
Filter by extent/size & Simplify	0.65 s	0.42 s	0.46 s	0.456 s

4 Discussion and Conclusion

This work evaluates optimization strategies for spatial join queries on client browser from distributed WFS servers. Our strategy is to automatically apply spatial filters based on map extent and feature size. The extent filter removes features beyond the currently viewed map while size filters remove features too small for the current zoom level. This kind of filters are suitable for the purpose of visual exploration. The results show the importance of spatial filtering in achieving acceptable query performance. As shown in Tables 3 and 4, the time for feature retrieval (11.6 s) dominates the time for spatial join and rendering if size filters are not applied. Even with size filters, the feature retrieval time (2.9 s) is still the largest component of the query time but at least it is within an acceptable range (3.5 s), which can be much lower if some or all of the queried data is cached. The results also show that naive implementation of spatial join (e.g. nested loop) scales poorly with the large number of features. Plane sweep, index nested-loop, and hierarchical traversal have similar performance, which makes plane-sweep a better choice due to its simplicity. Finally, the results show that geometry simplification can greatly reduce spatial join time, especially for features such as waterbody that can have tens of thousands of points in a geometry.

References

1 L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping based spatial join. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, pages 570–581, New York, August 1998.

2 T. Brinkho, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using rtrees. In *Proceedings of the ACM SIGMOD Conference*, pages 237–246, May 1993.

3 R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, Reading, MA, third edition edition, 2000.

4 P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

5 Cyrus Shahabi, Mohammad R. Kolahdouzan, and Maytham Safar. Alternative strategies for performing spatial joins on web sources. *Knowl. Inf. Syst.*, 6(3):290–314, May 2004.

6 J. Zhang, S. You, and L. Gruenwald. Towards GPU-accelerated Web-GIS for query-driven visual exploration. In *Proceedings of the 15th International Symposium on Web and Wireless Geographical Information Systems*, pages 119–136, Shanghai, China, May 2017.

7 T. Zhao, C. Zhang, and W. Li. Accessing distributed WFS data through a RDF query interface. In *Proceedings of GIScience*, 2016.

8 T. Zhao, C. Zhang, and W. Li. Adaptive and optimized RDF query interface for distributed WFS data. *International Journal of Geo-Information*, 6(4), 2017.