Efficient Interactive Visualization of Very Large Geospatial Query Results

Hannah Bast
University of Freiburg
Freiburg, Germany
bast@cs.uni-freiburg.de

Patrick Brosi
University of Freiburg
Freiburg, Germany
brosi@cs.uni-freiburg.de

Johannes Kalmbach
University of Freiburg
Freiburg, Germany
kalmbach@cs.uni-freiburg.de

Axel Lehmann
University of Freiburg
Freiburg, Germany
lehmann@cs.uni-freiburg.de

ABSTRACT

We present a web mapping application that offers interactive visualization of query results with hundreds of millions of geospatial objects. This is in contrast to existing applications, which are slow or unresponsive when the number of objects in the result is large. We describe a general technique, which works for any database engine that represents each geospatial object with a unique IDs and that can return a query result either with the objects or with the IDs. We have implemented a web mapping application using this technique and with the QLever SPARQL engine as backend. We evaluate it on queries on the complete OpenStreetMap (OSM) data, with result sizes ranging from small to very large. We compare it against the map interfaces of Overpass, PostGIS, and OSCAR.

CCS CONCEPTS

• **Information systems** → **Search interfaces**.

KEYWORDS

OpenStreetMap Data, Spatial Knowledge Graphs, Interactive Visualization, Data Exploration

ACM Reference Format:
Hannah Bast, Patrick Brosi, Johannes Kalmbach, and Axel Lehmann. 2023. Efficient Interactive Visualization of Very Large Geospatial Query Results. In *The 31st ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL '23), November 13–16, 2023, Hamburg, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3589132.3625594

1 INTRODUCTION

In a typical geospatial database, geospatial objects (points, open or closed polygons, or collections thereof) are annotated with additional data fields. The natural way to explore the results of queries on such a geospatial database is to display them on a map. Indeed, most query engines offer such map interfaces. However, these interfaces become slow or unresponsive already for a moderately large number of geospatial objects. In this work, we present *petrimaps*, a technique and web mapping application that allows interactive visualization (supporting the usual actions like zoom and pan) of hundreds of millions of geospatial objects. Our application can render heatmaps as well as the exact geometries, and it provides an export of the result as CSV, TSV, or GeoJSON. We have implemented our web application using the *QLever* SPARQL engine as backend and with the complete OpenStreetMap (OSM) dataset. An example for a geospatial SPARQL query and the visualization of the corresponding result using *petrimaps* can be seen in Figures 1 and 2.

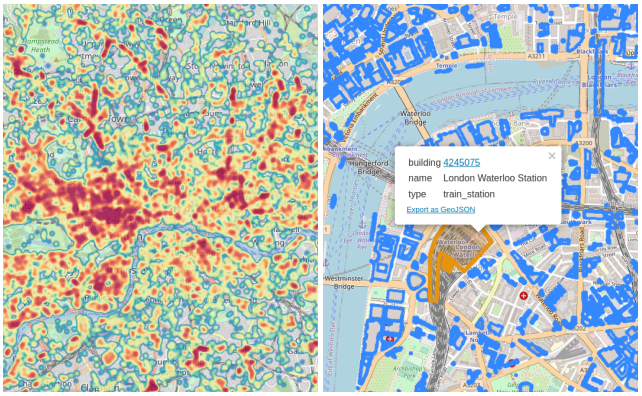

Figure 1: Results of the query *All named buildings with their id, name, and type* on a dataset built from the entire OpenStreetMap data (*planet.osm*), displayed by *petrimaps* as a heatmap (left) and with full geometries (right).

1.1 Contributions

We consider the following as our main contributions:

- We present a technique for the interactive visualization of query results with a very large number of geospatial objects on a map. The technique works for any database engine that represents each geospatial object using an internal ID and that can return a query result either with the objects or with the IDs.
- We provide an implementation of this technique using the *QLever* SPARQL engine as backend. The code is publicly available on https://github.com/ad-freiburg.
- We provide a complete web application, with the QLever UI for entering SPARQL queries, our web mapping application for showing the result on a map, and the complete OSM data; see https://qlever.cs.uni-freiburg.de/osm-planet.
- We evaluate the responsiveness of our web mapping application on queries with a variety of result sizes and compare it against the map interfaces of Overpass, PostGIS, and OSCAR.

SIGSPATIAL '23, November 13–16, 2023, Hamburg, Germany

ACM ISBN 979-8-4007-0168-9/23/11…$15.00
https://doi.org/10.1145/3589132.3625594

```
SELECT ?building ?name ?type ?geometry WHERE {
  ?building geo:hasGeometry ?geometry .
  ?building osmkey:name ?name .
  ?building osmkey:building ?type .
}
```

Figure 2: SPARQL query for example from Figure 1. The predicate *hasGeometry* from the GeoSPARQL standard connects entities to their geometry.

1.2 Related Work

We first introduce some approaches that combine the querying and visualization of geospatial data. *OSCAR* [1] is a tool that efficiently searches for prefixes, substrings, and geometric constraints on OSM data using a cell-based preprocessing. It has a UI that visualizes search results using client-side rendering. For low zoom levels, only one point per geometry is rendered, for high zoom levels the exact geometries are rendered.

The *Overpass API* (http://overpass-api.de/) is a tool to filter OSM data by tags or by a bounding box. It is most efficient if the result is constraint to a small bounding box [5]. It has a UI called *Overpass Turbo* (http://overpass-turbo.eu/). Like *OSCAR*, *Overpass Turbo* also visualizes all geometries as points on lower zoom levels, but renders these points as a heatmap. An example for a stand-alone visualization tool is *GeoServer* (https://geoserver.org/), a middleware that can render query results from a multitude of sources, for example from a PostGIS instance. However, *GeoServer* does not allow for arbitrary user-defined queries, but each query has to be specified by the maintainers of the server instance in advance. The following tools are used in combination with our tool *petrimaps*: *QLever* (https://qlever.cs.uni-freiburg.de, [3]) is a SPARQL engine for very large knowledge graphs which supports context-sensitive autocompletion of queries [4]. *osm2rdf* [2] is a tool that converts raw OSM data to RDF Turtle. This allows to load OSM into any SPARQL engine.

2 RENDERING QUERY RESULTS

A problem of existing approaches is that query results are usually fully transferred from the database to the client (usually a web browser) in some serialization format. The client then has to deserialize the objects, render them on a map, and manage potentially huge amounts of data during map interaction. For example, consider a simple serialization of Well-Known Text (WKT) geometries, separated by a line break. Then the query "all farms in OSM" already yields 28 MB of gzipped data[1]. But to render an interactive map, the client neither requires all objects, nor their exact geometries, nor any non-geometric data fields. It suffices to transfer only the objects in the bounding box corresponding to the current map view. Data fields for single objects can be transferred separately if they are explicitly requested by the user (for example, by clicking on an object). The client rendering load can be reduced further if the browser already receives a fully rendered bitmap, e.g., a PNG image.

We built an efficient middleware server which accepts a SPARQL query, sends it to the SPARQL engine, caches the results, and can deliver either a heatmap or an exact object map to the web client. The general architecture of our approach is shown in Figure 3 and

[1]https://qlever.cs.uni-freiburg.de/osm-planet/flQ7jQ → Download result as TSV.

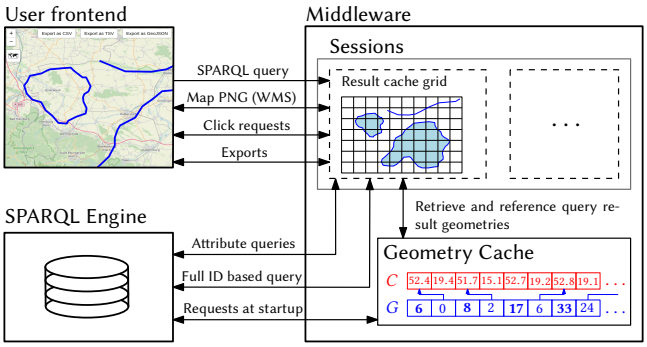

Figure 3: Architecture of our map frontend and the middleware backend. All map interaction is handled via the middleware backend, which also maintains a static cache of all geometries present in the database engine.

will be explained in detail below. Note that while we use the QLever SPARQL engine as the backend of our tool, our approach works in principle for all types of database systems and query engines that work by assigning a unique ID to each data item and that are able to expose those IDs to third-party software like our middleware.

2.1 Geometry Cache

To avoid having to transfer the full query results (in particular serialized geometries) from our SPARQL engine to the middleware, we cache the coordinates of each geospatial object stored in the SPARQL engine in a dense array C of coordinates. A second array G contains tuples `(qid, offset)`, where `qid` is the ID given to the geometry in the SPARQL engine, and `offset` is the position in C where the geometry of the object `qid` begins. G and C are loaded at startup by requesting all geospatial objects and their IDs from the SPARQL engine, ordered by the IDs.

2.1.1 Cache Compression. We compress the coordinate array C to C' as follows. All coordinates are scaled and rounded so that each of them can be represented as a 30-bit integer. We call the upper and lower 15 bits the *major* and *minor* part of the coordinate, respectively. Each part is stored in 16 bits, where the most significant bit is 1 for the major part and 0 for the minor part. For each coordinate of a geometry, we store both parts (in 32 bits) if the coordinate is the first of the geometry or if the major part is different from that of the previous coordinate. Otherwise, we store only the minor part (in 16 bits).

2.1.2 Retrieving Relevant Objects. When a SPARQL query arrives at the middleware, it is rewritten to request only the column with the geometry[2], in ID representation. The data reduction is substantial: for example, for the query "all farms in OSM", the compressed WKT literals have a size of 28 MB, whereas the ID representation requires only 2.6 MB. The array R of IDs is then sorted and intersected with G, yielding an array D of pairs `(row, gid)`, where `row` is the index of the geometry ID in R, and `gid` is its index in G. As both R and G are sorted, we use exponential search to speed up this intersection.

[2]For example, on the OSM RDF knowledge graph we can identify this column by looking for the *hasGeometry* predicate.

2.2 Map Rendering

We support two types of maps: raw maps, which simply render the point or polygon objects, and heatmaps, which depict object densities using color gradients. The latter is particularly useful for large query results. For both map styles, we implemented an untiled web map service (WMS) which can be box-queried and returns the requested map as a single PNG file. This section describes how the raw 2D array of the map's RGB values is filled.

2.2.1 Render Grids. To prepare the relevant objects for rendering, we insert points and polygons into separate coarse grids. For points, we directly store the `gid`, as the single coordinate can be quickly looked up in C. For open and closed polygons, we store the anchor points. Note that every user session uses its own grids. However, they are reused for subsequent requests during map interaction. Building these grids has multiple benefits: First, we can use the grids to directly render maps of the result set if the desired ground resolution is larger than the grid cell length. Second, for smaller ground resolutions, the grids provide efficient access to the parts of the result set required for rendering. Third, for the polygon anchor points, the grid allows for a simple delta encoding of the geometry coordinates: we don't store raw coordinates, but their offsets to the upper right corner of the grid cell. These are scaled to the $[0, 256]$ range to fit in an unsigned 8 bit integer. We use a grid cell size of $c = 65,536$ web mercator units (~65 km at the equator). This means that our scaled 8 bit integer offset coordinates are precise enough until a ground resolution of $d = 256$ web mercator units per pixel. To also support lower ground resolutions, we additionally maintain an auxiliary grid containing only the geometry IDs for polygons. If the requested resolution is below 256 map units per pixel, we access this grid and retrieve the exact geometries from G.

2.2.2 Heatmap Rendering. For rendering the heatmap, we maintain a two-dimensional array H containing an integer for each pixel of the requested map. The value is increased by 1 if an element is covering the pixel. As mentioned above, we can populate H from the precomputed grids for resolutions $\geq c$. To achieve this, we simply iterate over the grid cells in the requested bounding box and increase the corresponding pixel value in H by the number of elements stored in the cell. If the requested resolution is between c and d, we iterate over all coordinates in the active cells and increase their corresponding heatmap pixel value by 1. If the requested resolution is below d, we do the same, but using the exact geometries obtained via the auxiliary grid, as described above. For each element (x, y) in H, a 2D intensity *stamp* (with intensities decreasing from the center) is then printed to the final map drawing (again a 2D pixel array M), weighted by $H[x][y]$. Pixel colors in M are based on this intensity value. We use *libheatmap*[3] to generate M and *libpng*[4] to convert M into a PNG image.

As we only add polygon anchor points to the heatmap, they may not appear connected on small ground resolutions (Fig. 4). To avoid this, we densify them when the geometry cache is filled, using a distance between added anchor points high enough so that polygons are rendered without gaps for resolutions $\geq d$. For resolutions $< d$, we densify the anchor points again on the fly.

[3]https://github.com/lucasb-eyer/libheatmap
[4]http://www.libpng.org/

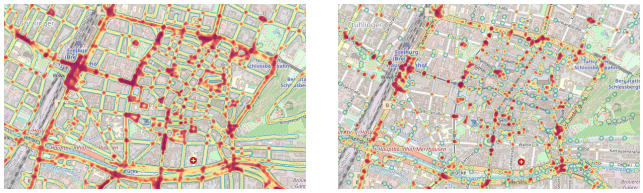

Figure 4: Heatmap of a query for all streets in Freiburg, with (left) and without (right) polygon densification.

2.2.3 Raw Object Rendering. To render maps containing the raw objects, we essentially use the same approach as for rendering heat maps. However, we use only a single stamp color and use only binary object counting (a pixel is either occupied, or it isn't).

2.3 Map Interaction

Each SPARQL request spawns a session in our middleware, maintaining its own result grids during user interaction to allow fast re-rendering of the current map extent. The downside of this approach is a high memory consumption in our middleware for sessions showing large result sets.

The maps produced above are static PNG images. So far, we have not transferred any columns except the internal geometry IDs. To provide efficient interactive access to the query result columns, we again use the grids built for the query result rendering. If a user clicks into the map, we retrieve the nearest object from the grids and explicitly request the corresponding row from the SPARQL backend[5]. The object geometry is then simplified according to the current map resolution, and returned to the client as a GeoJSON object. The client renders this object in a highlight color, and also shows a popup containing the data columns (Fig. 1, right).

Note that we did not distinguish between open and closed polygons added to the grid in Section 2.2.1. To do so, we add a special marker coordinate to C at the end of each closed polygon. For such polygons, we then use point-in-polygon tests on user interaction.

3 PERFORMANCE EVALUATION

We formulated equivalent queries in 4 languages: (1) SQL queries against a PostGIS database filled with OSM data using osm2pgsql (with appropriate indexes), (2) Overpass API queries against an exclusive Overpass API instance, (3) search engine queries against OSCAR, and (4) SPARQL queries against QLever, filled with RDF data for OSM produced by osm2rdf. Results were displayed for (1) in the standard web GUI for PostgreSQL databases (pgAdmin), for (2) in Overpass Turbo, for (3) in the OSCAR web GUI, and for (4) in our own tool petrimaps. We used the following queries:

(**Q1**) The single building with OSM way id 98284318
(**Q2**) All castle buildings (`building=castle`)
(**Q3**) All train station buildings (`building=train_station`)
(**Q4**) All farm buildings (`building=farm`)
(**Q5**) All residential buildings (`building=residential`)
(**Q6**) All streets (`highway=*`)

[5]Row selection is achieved by a combination of `OFFSET <row>` and `LIMIT 1`. This is very fast if the SPARQL engine has cached the query result in the ID space and thus doesn't have to process the query again. For backends that don't provide such a cache we could also initially transmit all the columns of the result to our midddleware as IDs and store them. Selecting a row would then be implemented by asking the backend to resolve the IDs of the corresponding row to their respective serialization.

The evaluation of the map rendering times for (1), (2), and (3) was tedious, which is why we only evaluated 6 queries. However, these times mostly depend on the total number of result objects (and their average number of geometry anchor points), not on the distribution of geometries over the planet, or the query structure. We hence chose queries representative for different result set sizes.

In Table 1 we measured the database retrieval time (t_R) and the time between the successful retrieval of the query data and the finished rendered map (t_M). For petrimaps, this includes the transfer time of the rendered PNG. In Table 2 we measured the total blocking time (TBT, the total time the browser was not responsive for more than 50ms, excluding the first 50ms) incurred by increasing the zoom level by 1 (t_Z), and the TBT incurred by a map pan from center to right (t_P). TBT was measured using Google Lighthouse[6].

Client-side experiments were run on an machine with 32 GB of RAM. The petrimaps middleware, Overpass, and PostgreSQL were run on a machine with 256 GB of RAM. Filling the geometry cache for petrimaps took around 2 hours. For OSCAR, we used the official instance[7]. Our complete evaluation setup can be found online[8].

Table 1: Efficiency of the web GUIs for PostGIS (pgAdmin), Overpass (Overpass Turbo), OSCAR, and *petrimaps* for our test queries. $|R|$ is the result size, t_R the time until all data has been received from the database, and t_M is the initial map rendering time. A time of × means the application crashed during rendering, or the query was cancelled due to its size.

		Overpass		pgAdmin		OSCAR		petrimaps	
	$\|R\|$	t_R	t_M	t_R	t_M	t_R	t_M	t_R	t_M
Q1	1	80ms	40ms	0.1s	20ms	0.1s	0ms	140ms	63ms
Q2	2.5k	6.7s	2.1s	0.5s	1.1s	0.1s	0.2s	50ms	0.3s
Q3	48k	1.5m	32.3s	2.6s	3.5s	0.5s	0.7s	0.1s	0.4s
Q4	358k	3.5m	×	3.8s	×	1.5s	2.4s	0.3s	0.4s
Q5	15M	×	×	41s	×	×	×	3.1s	4.2s
Q6	218M	×	×	8.5m	×	×	×	32s	58s

None of the other GUIs were able to display the results for Q5 and Q6, and both Overpass and pgAdmin also failed to render Q4. For pgAdmin, the reason was a hardcoded restriction to 100,000 elements. Overpass either crashed the browser while trying to render the results, or the backend aborted the query. OSCAR aborted *Q5* and *Q6* because of the result set size. For all but Q1 and Q2, petrimaps had the fastest map rendering time. The slower initial map rendering time for Q1 can be explained by the overhead of first building the session grid and rendering / transferring a full PNG image. All other GUIs directly rendered the retrieved geometry on the client side. The faster initial map rendering time of OSCAR for Q2 can additionally be explained by the fact that OSCAR only considers a single anchor point per geometry at the zoom level required for this map. The more objects we draw, the faster petrimaps becomes relative to to other tools.

Overpass Turbo, pgAdmin and Oscar quickly reached interaction TBTs of over 600 ms, which is the worst (red) category in Lighthouse scores and is interpreted as an unresponsive application. In contrast, petrimaps stayed responsive and usable for all queries.

[6]https://developer.chrome.com/docs/lighthouse/performance
[7]https://www.oscar-web.de
[8]https://qlever.cs.uni-freiburg.de/mapui-petri/evaluation/

As petrimaps has to render and transfer a new PNG image after each map interaction, we additionally measured the time required to render and transfer the map image after each interaction (t_Z^*, t_P^*). These times were below 300 ms for all queries and all interactions.

For the entire OSM data, we required 69 GB of RAM to hold the geometry cache. The largest of our queries, Q6, required 6 GB of RAM to hold the session's result cache grid.

Table 2: Responsiveness of pgAdmin, Overpass Turbo, OSCAR, and *petrimaps* for our test queries. t_Z is the total blocking time (TBT) for a single zoom-in. t_P is the TBT for a single map pan from center to right. A time of — means the application could not display the results. For petrimaps, t_Z^* and t_P^* give the time required to render and transfer the map PNG.

	Overpass		pgAdmin		OSCAR		petrimaps			
	t_Z	t_P	t_Z	t_P	t_Z	t_P	t_Z	t_Z^*	t_P	t_P^*
Q1	0ms	0ms	0ms	0ms	0.1s	10ms	0ms	53ms	0ms	66ms
Q2	0.2s	30ms	0.1s	20ms	0.1s	70ms	0.1s	0.2s	10ms	0.1s
Q3	2.9s	2.3s	1.2s	0.3s	0.3s	80ms	80ms	0.3s	0ms	0.1s
Q4	—	—	—	—	0.7s	1.2s	60ms	0.2s	0ms	0.1s
Q5	—	—	—	—	—	—	90ms	0.2s	0ms	0.1s
Q6	—	—	—	—	—	—	0.1s	0.3s	10ms	0.1s

4 CONCLUSIONS AND FUTURE WORK

We have presented *petrimaps*, a web mapping application for the interactive visualization of query results with hundreds of millions of geospatial objects. We have compared it to the web interfaces of Overpass, PostGIS, and OSCAR, which become slow or unresponsive when the number of objects gets larger. Our code is publicly available on https://github.com/ad-freiburg and a live version of our web application is available on https://qlever.cs.uni-freiburg.de/osm-planet.

As of this writing, our implementation caches a pre-processed representation of *all* geometries of the given dataset. For the complete OpenStreetMap data, the initial pre-processing takes around 2 hours. The result is serialized to disk, so that when the application is restarted, it can be read back into RAM in around 2 minutes. We plan to implement the following improvements. First, lazily pre-process the geometries in the background and when queries arrive (then pre-processing the geometries for that query), so that the application is responsive right from the start. Second, reduce RAM usage by loading only those pre-processed geometries needed for the current query. Finally, we consider integrating this whole mechanism into the QLever SPARQL engine.

REFERENCES

[1] Daniel Bahrdt, Stefan Funke, Rick Gelhausen, and Sabine Storandt. 2017. Searching OSM Planet with Context-Aware Spatial Relations. In *GIS 2017*.
[2] Hannah Bast, Patrick Brosi, Johannes Kalmbach, and Axel Lehmann. 2021. An Efficient RDF Converter and SPARQL Endpoint for the Complete OpenStreetMap Data. In *GIS*. ACM, 536–539.
[3] Hannah Bast and Björn Buchhold. 2017. QLever: A Query Engine for Efficient SPARQL+Text Search. In *CIKM 2017, Singapore, November 06 - 10*.
[4] Hannah Bast, Johannes Kalmbach, Theresa Klumpp, Florian Kramer, and Niklas Schnelle. 2022. Efficient and Effective SPARQL Autocompletion on Very Large Knowledge Graphs. In *CIKM*. ACM, 2893–2902.
[5] Roland M. Olbricht. 2015. Data Retrieval for Small Spatial Regions in OpenStreetMap. In *OpenStreetMap in GIScience*. Springer, 101–122.