e-Sight: Real-time cloud platform for visualizing edge transport infrastructure information

Marco Jansen, Fatjon Seraj

Pervasive Systems Group

University of Twente, Enschede, The Netherlands

Email: 345864@student.saxion.nl, f.seraj@utwente.nl

Abstract—The huge amount of streaming information generated by the new wave of edge devices that are used to monitor a plethora of everyday aspects, prompts the need for efficient techniques to handle, process, aggregate, and visualize this stream of data. One such field is the continuous transport infrastructure monitoring, where smartphones inside driving vehicles act as edge sensing and computing nodes to measure the quality of the road pavement among other things. Although the location accuracy of such devices is within the acceptable bounds, the accumulated error can lead to large deviations from the location of interest, reducing the measurement credibility. The data represent a geographically vast infrastructure network in need of real-time, bird-eye visualization. This paper describes the implementation of such a real-time platform and the challenges the task provides. By implementing relatively new open-source cross-platform environments, the platform is scalable and versatile, reducing the costs associated with cloud-based implementation systems.

Index Terms—Crowd-sensing, predictive maintenance, infrastructure health monitoring, computational geometry, Delaunay Triangulation, map-matching, map generation

I. INTRODUCTION

Nowadays, sensors are becoming becoming the synonym of ubiquitous. Recent years we witnessed a growth of cheap sensors, especially sensors found in smart devices like smartphones, smartwatches etc. All these sensors generate large amounts of data with a considerable margin of error. On the other hand, smartphones nowadays have access to their location through GNSS services like GPS, Glonass, Galileo. This provides the opportunity to couple the sensor data with the location retrieved by the GNSS, representing a spatial information of the site where the data is generated. One particular example of smartphones sensors like the accelerometer and gyroscope coupled with geo-location information is the measurement of road geometry parameter [1].

When measuring roads, an application collects sensor values in a buffer and applies filters to enhance the useful part of the signal and reduce the noise. Then it extracts features from this buffer and uses these features to calculate an index which can indicate the roughness or pinpoint road anomalies. However, one single drive over the road segment cannot provide the full infrastructural state of the segment when measured with the sensors found in smartphones. These measurements become more accurate when combined with results from multiple phones on the same road segment.

The Netherlands has a registered fleet of 8.222.974 motor vehicles as per 2017 report of the dutch Central Bureau of Statistics [2]. If considering only one percent of these vehicle passengers would use their smartphone to measure the road quality, there would be about 82.230 active streams of road measurements connected to the cloud service. This would result in a strain on the back-end, which will flooded with torrents of measurements and locations. To increase the accuracy multiple measurements on the same location can be aggregated into road segments, that later are used to visualize the aggregated data.

The challenges in proceeding this path, consist in finding the optimal solutions for a real-time storing, aggregating and visualizing system. What database technology is suitable to store streaming spatial information so that the results can be queried efficiently? Another crucial challenge is keeping the application real-time, so results can be processed as they come in and shown immediately, making the information useful for a multitude of purposes, Visualization becomes challenging when the data is growing larger. Loading a large amount of road segments represented by points increases the complexity which can make the map unresponsive or cause crashes. Decisions should be made how to efficiently query only the points specific to the location of interest, considering that the user will drag the screen all over the road of interest. Another problem is the efficiency of processing measurements. Multiple clients will be sending results and this should be processed as fast as possible to keep the application real-time. The processing consists of aggregating the measurements from multiple clients into road segments with an average index which indicates the road quality for this segment.

This paper presents an implementation of [3] where real-time crowd-based infrastructure measurements are aggregated and visualized using Delaunay Triangulation (DT) for creating and updating infrastructure segments.

In this paper we describe a real implementation of the aggregation and visualization system to deal with the challenges stated above. Section II gives a brief overview of existing works related to aggregation and visualization of spatial data. Section III outlines the methodological approach used to the challenges stated above. The results will be evaluated in Section IV. Concluding remarks, open issues and future work are discussed in Section V.
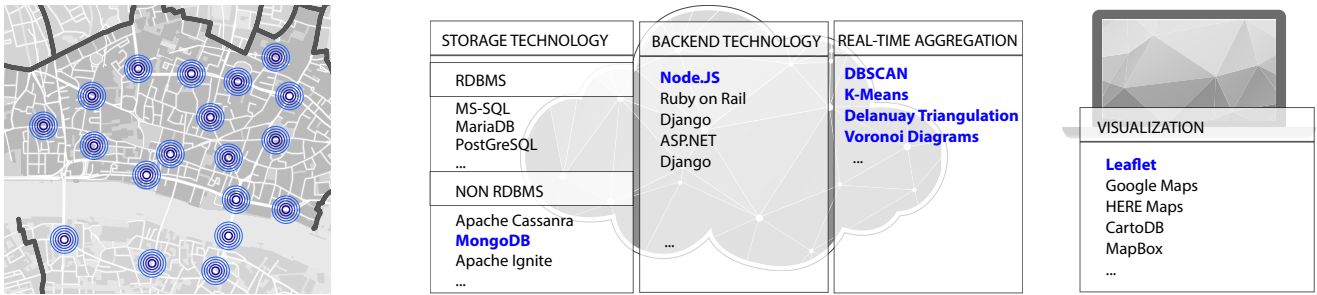
Fig. 1. system overview and candidate technologies

II. RELATED WORK

Due to the fact that crowd-based large scale monitoring is a relatively a new domain not many works considered building standalone back-end support for their applications. Mostly these applications are deployed and rely on pricey cloud computing platforms such Amazon AWS, Microsoft Azure, Google Cloud etc.

To the best of our knowledge only SmartRoadSense [4] is the closest similar project which collects measurements of road roughness from smartphones and processes them periodically in their indigenous back-end platform. All the measurements are stored in a PostgreSQL database with PostGIS extensions. The measured point locations are pulled to the nearest road and averaged on a certain radius. The averaged results are uploaded once per day after the daily processing to the CartoDB server. The visualization is handled by CartoDB by updating the map with the latest uploaded results [4]. One downside of this approach is that it isn't real-time, processing only occurs periodically (once per day in this case). This method implicates a large computational load during the processing phase. Another downside concerns to the stretching of measurements to the nearest road, there is a chance that the presence of several lanes is neglected in this approach. Considering that the system relies on information of existing mapping services, it cannot foresee the case when the vehicle is driving on a new road which isn't present on the mapping service. One last minor issue is delay on measurement publication, the results will be available the day after, thus limiting the usability of the information for other purposes.

CartoDB is location intelligence software which offers ways to visualize spatial business data. CartoDB can be used to upload your spatial business data and generate a variety of customize-able maps. The developers of CartoFB describe their platform as: "A one-stop shop of geospatial tools, services, and APIs for discovering and predicting the key insights from your location data, CARTO Engine empowers your organization with scalable analysis and enrichment solutions you can fully embed on your web and mobile apps" [5]. Although CartoDB is a very powerful tool, it does not offer functionality for real-time clustering and visualization. It is possible to sync tables every hour, but not on every result that comes in. Requires more development on the back-end side of the system to handle the real-time aggregation of the incoming data. However still the transmission of the aggregated results should be sent and integrated with the CartoDB server, than the new maps would be generate on hourly bases. Some CartoDB functionality are restricted to the enterprise edition, thus becoming more expensive in the long run.

Mapbox [6] has a series of products related to mapping. They offer products to create custom maps, to do navigation and Mapbox GL JS. Mapbox GL JS is a JavaScript library that uses WebGL to render interactive maps from vector tiles and Mapbox styles. It can be used load location data like GeoJSON from sources and display it on a map. Another interesting product is the Mapbox Uploads API. This API can be used to upload spatial file types such as GeoJSON, KML, GPX, Shapefiles and CSV. It processes the data into raster tiles or Mapbox vector tiles readable by Mapbox GL and Mapbox js. However, these products don't have solutions for large real-time changing data sets. For large data sets it recommends tiling on the server, which is hard to implement for real-time changing data sets. This implementation would require regenerating a lot of tiles on multiple zoom levels when the data set changes slightly. These tiles could be real-time generated using vector tiles, but this approach can be complex and requires some extra processing and caching when possible.

III. METHODOLOGY

The implementation consists of an Android application that serves as an edge computing device, that buffers the smartphone sensors values to feed the information into a machine learning model. The model among other things, calculates the road quality indicators comparable to International Roughness Index (IRI). After IRI calculations, the results are coupled with the geo-locations of the measured segment and send to the cloud back-end system where the data is aggregated and stored. The challenge consists on choosing the most appropriate back-end system technology, database technology and visualization system sa well as combining them into a unified product able to give a real-time bird eye view of the state of the infrastructure. An overview and candidate technologies are shown in Figure 1.

A. Database

From the plethora of existing database technologies that can store data efficiently, only the ones with spatial query support are of interest. The spatial data handling capabilities

are paramount in this research, because it has to store and process sparse geo-location information for aggregation and visualization. We identified two well-known mature databases with spatial query support. PostGreSQL with the PostGIS extension and MongoDB. The biggest difference between these databases is their method of storing the information. PostGres is a RDMS (Relational Database Management System) database technology that uses SQL (Structured Query Language) to manipulate the data stored in a predefined structured schem. Whereas MongoDB is a nonSQL document-oriented database that uses JSON documents with schema. Both these database systems are free to use and offer spatial query support, where as PostGIS offers more spatial query operations.

TABLE I
COMPARISON MONGODB VS POSTGRESQL/POSTGIS

Database	Pros	Cons
MongoDB	Speed, Scalability, Flexibility Geographically Distributed Clusters Tiered Storage	NoSQL
PostGreSQL/ PostGIS	Big variety spatial queries, SQL	Scalability, Speed Proprietary Extensions Legacy Relational Overhead

According to a published performance analysis MongoDB performs faster with spatial queries with and without indexes. The performance is important for the ability to aggregate new measurements real-time. [7]

Based on the pros and cons as seen in Table I we've chosen to go with MongoDB over PostGreSQL with PostGIS.

Spatial data is stored in GeoJSON format and indexed with the MongoDB *2dsphere* index. A 2dsphere index supports queries that calculate geometries on an earth-like sphere. 2dsphere index supports all MongoDB geospatial queries: queries for inclusion, intersection and proximity. For more information on geospatial queries [8]. GeoJSON is a geospatial data interchange format based on JavaScript Object Notation (JSON) that represent data about geographic features, their properties, and their spatial extents [9].

There are two important spatial operators in MongoDB which were used in the implementation.

1) *$near* operator which can be used to find points near a given point and sorts the results based on distance from the given point. This query works with a *$maxDistance* operator which is the radius in metres. This query can be used to find the nearest road segment within a certain radius when a new measurement with measurement comes in.

2) *$geoWithin* operator can be used to find all road segments within a certain area or box. The results are not sorted with this operator and can be used for example to find retrieve all road segments in the view-port of the user in the visualization.

B. Back-end system

The back-end is written in JavaScript using Node.js with Express. "Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js package ecosystem, *npm*, is the largest ecosystem of open source libraries in the world [10]." The node package manager (*npm*) is the package manager for JavaScript and the world's largest software registry. It can be used to discover packages of reusable code [11]. Reusable modules can save development time by not *"reinventing the wheel"* and increase the quality of the product when the module is of good quality. The major drawback of **code reuse** is that the developer is bound to and dependent to the module architecture and how it's supposed to be used. Another drawback is the quality of the code, the quality of the code could be lower than set standards and existing bugs in the code get replicated. The source code can be checked, but understanding and reading the code can easily becom a e tedious process.
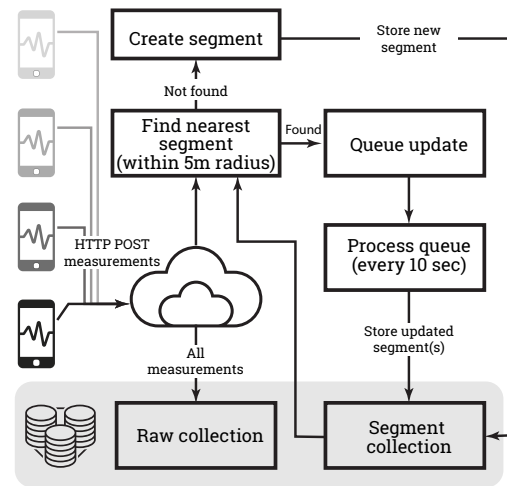

Fig. 2. Back-end system receive

The server exposes a set of HTTP APIs that can be used for sending results and retrieving road segments. 2 A result consists of a index which is a number and the location coordinates in {Latitude, Longitude} format. The processing of an incoming measurement is shown in Fig. 2. Smartphones send measurements in a HTTP post request in JSON-format. On the back-end a near query will be executed to find the nearest segment within a 5 metres radius. A segment will immediately be created if there are no results. If a segment was found an update will be queued which is processed later. The processing of the queue is executed every 10 seconds. Updates within the same cluster are grouped together in the queue. The queue itself is a MongoDB collection. The MongoDB internal locking takes care of the concurrency issues when creating segment and adding items to the queue. It was not possible to find and update a segment using existing fields from the stored document in one query. If this was done in multiple queries concurrency issues would occur. This was the reason why the queue implementation was used.

To implement the DT a few JavaScript modules were tested from the *npm* library. The modules that were tested are Delaunator [12], incremental-delaunay [13] and d3-voronoi [14].
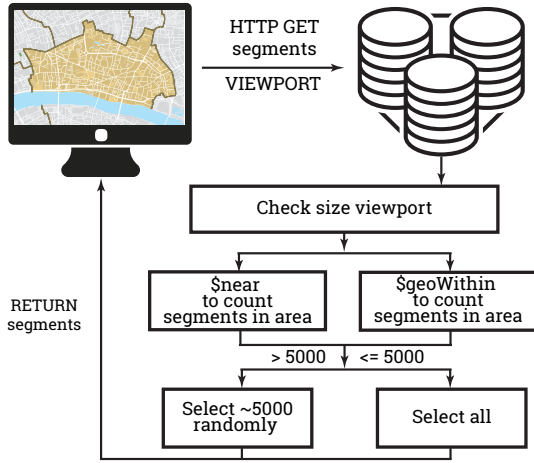
Fig. 3. Back-end system retrieve

The Delaunator is fast in creating the triangulation from a data set, but does not provide functions to retrieve the nearest neighbour or to build the triangulation incremental.

Incremental-delaunay provides the required incremental functionality, but was already aged and no longer compatible and non functional.

The d3-voronoi library does offer a find function to search the nearest site to a point, but does not have the functionality to build the triangulation incrementally. We tried recreating the entire triangulation every time a new point was received, but the procedure was inefficient and sluggish requiring a few minutes to create a triangulation out of a few thousand points on the test system.

None of these JavaScript modules could offer the incremental build functionality we required. Instead we opted for a C++ implementation, namely the CGAL library. CGAL is a software project that provides easy access to efficient and reliable geometric algorithms in the form of a C++ library, such triangulations, Voronoi diagrams, surface and volume mesh generation, geometry processing, convex hull algorithms, KD trees etc.. CGAL is used in various areas needing geometric computation, such as geographic information systems, computer aided design, molecular biology, medical imaging, computer graphics, and robotics [15]. This implementation loads all points from the database and creates a DT incrementally. The nearest vertex is searched for every point and the distance is calculated. If the distance is smaller than 5 metres it will merge with the existing vertex and update the triangulation. When the distance is smaller than 5 metres a new vertex is inserted. In order for us to use CGAL library as active aggregation algorithm, we coded a single threaded implementation to link with Node.js.

C. Visualization

Geovisualization of the results on a map can be handled by multiple services. Options include Google Maps, Mapbox, Leaflet, OpenLayers, CartoDB, Here Maps, ArcGIS Online etc. All these platforms offer ways to show and manipulate

geo-points on a map. The biggest differences are in the price, API, speed and backend technology, Table II.

TABLE II
COMPARISON MAPPING TOOLS

Platform	Free	Speed	Syntax
Google Maps	Limited	Good	Easy
Mapbox	Limited	Good	Easy
Leaflet	Yes	Good	Easy
OpenLayers	Yes	Decent	Difficult
CartoDB	No	Good	Easy
Here Maps	Limited	Good	Easy

Based on the comparison Leaflet, a javascript library for interactive maps, has been chosen for this implementation. The view-port gets passed to the back-end when requesting for the road segments. The center, bounds and distance from center to bounds are passed to the server. All these values are retrieved or calculated using leaflet functionality.

Using the view-port, the back-end can search for segments in the database inside this view-port using the geoWithin operator or the near operator. The bounds are important for using geoWithin and the center with distance to one of the bounds for the near query. The entire process of retrieving segments is shown in Fig. 3.

It is not practical to show all the segments at once on the map, because the *out of memory* errors as well as navigation latency with the map interface becoming unresponsive. The implementation ensures that only the segments within the area displayed will be queried. However, this might still cause *out of memory* errors if the road network in the area is too dense and the amount of data points is huge, as the case of urban centers. The problem is tackled by showing only the amount of points in which the front-end still navigates smoothly. On the testing system this was around 5.000 points. Limiting the query to 5.000 points resulted in seeing only a part of the area in the way it was stored. This will result in a very dense small area being shown. To make sure these results are spread across the area a random field has been added on the creation of a road segment. This field is indexed and can be queried efficiently. Using this random field it is possible to return around 5.000 points using the total amount of points to determine which random field value was needed in order to be returned to the client. On lower zoom levels this would result in showing all available segments. On higher zoom levels it a bird eye overview of state of the infrastructure.

Filters are available which can be used to only show certain points where the index is in the chosen range. This results in only querying points with an index in this range.

D. Concurrency

Real-time aggregation of the incoming measurements can cause concurrency issues. A race condition occurs when two or more phones in the same area, while uploading results, try to access the shared geo-location segment on the database and update it simultaneously. Because the location has to be updated consecutively and both phones locked the location for update, the order in which the phones will attempt to access the

shared location is unknown. Therefore, the update of that geo-location segment is dependent on the scheduling algorithm, i.e. both phones are *racing* to access/change the data.

To solve these concurrency issues we rely on MongoDB ability to handle the geo-locking. However, this was not always possible when operations were split into multiple queries. When finding an existing segment and updating it, it was possible that other processes took the lock in between and retrieved the document before it was updated. This can cause race conditions when this happens on documents which are found multiple times before the update was done. It was not possible to execute a find and update or create using one query. There is functionality for finding and modifying in one query, but when doing this you are unable to retrieve existing fields of this document. We required the existing fields in order to calculate the new field using the new results. It was possible to execute a *find* or *create* functionality in one query with MongoDB and this left us with only having the problem for updating segments real-time. To solve this for updating the segments we have implemented a queue in which the updates were queued. This queue was processed periodically and made sure updates which applied the same segment were grouped together in order to prevent concurrency issues.

IV. EVALUATION

The back-end is set up to run multiple processes based on the amount of available cores. On the test system there were 4 physical cores and 8 available cores with hyper threading. The specifications of the test system are as follow: MacBook Pro 15'(2013),2,3 GHz Intel Core i7, 16 GB 1600 MHz DDR3, 500 GB SSD .

OpenStreetMaps was used for GPS-traces with randomized indexes for test data sets. A JavaScript program has been written which sends these results to the server in chunks to simulate a real world situation. It was set up to send chunks of 100 measurements every 20 ms. This is 5.000 points per second. At this rate there was a small delay shortly after the updates were executed periodically, but it caught up soon after. We've also been tested 10.000 per second, but this resulted in freezes and very delayed responses after a while. The exact amount of points the implementation can handle on the test was not measured, but is believed to be between 5.000 and 10.000 for the test system. In a real world scenario where the servers are scalable the results will be better. The exact amount of points the test system can handle is not of interest.

For the sake of comparison, the system is also tested on a Raspberry Pi 3B running Raspbian OS, (Figure 5 shows the map from a system running in a RaspberryPi 3B). The amount of points this system handle is a lot lower than the test machine. The CPU usage was recorded with different rates of measurements coming in. On the Raspberry delays start to build up when more than 60 results per second are received. However, this is not due to CPU capabilities, rather due to the SDcard lower write/read speeds. The most recent version of MongoDB is not build for 32-bit operating systems and we had to use an older version (2.4.1) of MongoDB. This

also implies using an older version of the Node.js, MongoDB native driver (2.2) and adjusting one line of code that handles the connection. The CPU usage with different measurement rates is shown in Table III.

TABLE III
CPU USAGE

Points per second	CPU usage Node	CPU usage MongoDB
Raspberry (CPU 400% max)		
40	5%	20%
50	9%	21%
60	10%	23%
Test system (CPU 800% max)		
4000	58%	230%
5000	71%	260%
6000	89%	330%

To compare the performance of the MongoDB *near* operator in a query versus the *geoWithin* operator a few experiments are conducted. Both the options are implemented and executed on different zoom levels, which resulted in a different amount of points in the area. The data is plotted in Fig. 4. It clearly shows that the *near* operator works better with fewer points and the *geoWithin* works better on larger amount of points. This can be explained due to the fact that the *near* operator uses internal sorting. On higher zoom levels there will be more points which can be shown. For this purpose the *geoWithin* query should offer the best solution unless the data set is small. For lower zoom levels where there are less points the *near* query will outperform the *geoWithin* query. This is most likely due to better use of the *2dsphere* index. The results are from executing the query and counting the total amount of points returned. After the counting around 5.000 results were randomly selected and returned by the implementation. The 2dsphere index was present in both tests.
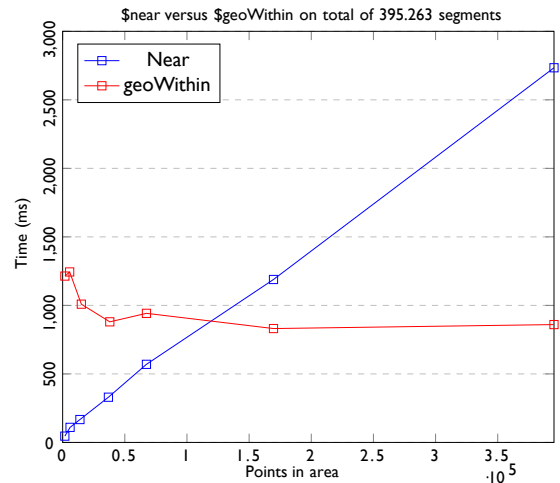

Fig. 4. *near* vs. *geoWithin*

The DT has been tested on three different data sets with a different amount of points to measure the performance and memory usage. All the points were retrieved from a MongoDB collection and then inserted into the DT using the methodology described in the back-end system subsection. All tests were executed on a single thread. The memory usage was measured

using xCode debug tools. The results of this tests can be seen in Table IV.

TABLE IV
RESULTS DELAUNAY TRIANGULATION (DT) SINGLE THREADED

Amount of points	Time (sec)	Memory used	Points per second
688.965	317	77.5 MB	2173
152.589	65	28 MB	2347
61.339	25	17.5 MB	2453

There is room for improvement in the current implementation. Errors in the GPS values can occur. Wrong locations cause segments to be created on the wrong positions. To remove errors from the data set periodical checks can be executed which check the last update date and the amount of processed measurements into a segment. If a segment only contains a few processed points and was last updated a while ago it could be considered as an error. Ways to determine whether a segment was a GPS error can still be researched and implemented.

Retrieving the segments is done using the MongoDB *near* and *geoWithin* operators. To decide what segments are returned, the amount of segments in the area is counted first. This could be optimized by estimating the total count. Another way to optimize the response times could be finding the optimal *viewport* distance to choose the near or *geoWithin* query.

A series of zooms executed on the front-end trigger multiple requests to the back-end system. This causes unnecessary load. Series of zooms could be detected and only the last request could be sent in that case. Leaflet does not offer functionality to detect series of zooms, this will have to be detected and implemented manually.

The implementation weights all results equally into the segments. The response time for maintenance is proportionate to the duration it was measured before the maintenance. After road maintenance the interest are results after the maintenance. A possible way to solve this can be only taking in consideration the most recent results and creating an UI for data manipulation. This UI could make it possible to generate results from other periods using the raw data collection. These older generated results could then be cached for faster access.

V. CONCLUSION

Currently the system is specific to measurements from phones and aggregates the measurements in a predefined way. This implementation is used for real time aggregation and visualization of streaming transport infrastructure. Querying the system and retrieving segment information from the cloud is fast and the most recent segments are retrieved. The cloud system easily handles large amounts of incoming measurements simultaneously. On the test system this is around 5.000 measurements per second. The DT implementation in C++ is very promising and can handle around 2.200 locations per second on a single thread. The performance on multiple threads would be a lot higher and could possibly surpass the active clustering algorithm used by MongoDB spatial queries.

Fig. 5. Map showing a road segment with calculated DT and convex hull in street zoom level

In a real world scenario where processing power is scalable the results will be higher. Creation of segments is real-time in this implementation and the updates are done periodically to avoid concurrency issues.

In the future this implementation will be adjusted to create a real-time spatial data analysis platform to make it multiple-purposes research and visualization platform. This platform will be used for real-time crowd monitoring, asset tracking and monitoring, transport and infrastructure logistics, animal monitoring for wildlife preservation, animal migration etc..

REFERENCES

[1] F. Seraj, B. van der Zwaag, A. Dilo, T. Luarasi, and P. Havinga, *RoADS: A road pavement monitoring system for anomaly detection using smart phones*, ser. Lecture Notes in Computer Science. Springer, 1 2016, no. 9546, pp. 128–146.

[2] C. B. voor de Statistiek, "Cbs statline - motorvoertuigenpark; inwoners, type, regio, 1 januari," http://statline.cbs.nl/StatWeb/publication/.

[3] F. Seraj, N. Meratnia, and P. J. M. Havinga, "An aggregation and visualization technique for crowd-sourced continuous monitoring of transport infrastructures," in *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, March 2017, pp. 219–224.

[4] G. Alessandroni, L. Klopfenstein, S. Delpriori, M. Dromedari, G. Luchetti, B. Paolini, A. Seraghiti, E. Lattanzi, V. Freschi, A. Carini, and A. Bogliolo, "Smartroadsense: Collaborative road surface condition monitoring," in *Proceedings of the UBICOMM 2014: The Eighth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. IARIA, 2014, pp. 210–215.

[5] Carto, "Location intelligence platform and apis," https://carto.com/engine/.

[6] "Mapbox documentation," https://www.mapbox.com/documentation/.

[7] K. S. R. Sarthak Agarwal, "Performance analysis of mongodb versus postgis/postgresql databases for line intersection and point containment spatial queries," *Korean Spatial Information Society*, vol. 24, pp. 671–677, 2016.

[8] MongoDB, "MongoDB Documentation," https://docs.mongodb.com/.

[9] D. M. D. A. G. S. H. S. Butler, H. and T. Schaub, "The geojson format," *RFC 7946*, 2016.

[10] Node.js, "Node.js," https://nodejs.org/en/.

[11] npm, "npm," https://www.npmjs.com.

[12] V. Agafonkin, "Delaunator," https://github.com/mapbox/delaunator, 2017.

[13] M. Lysenko, "incremental-delaunay," https://github.com/mikolalysenko/incremental-delaunay, 2013.

[14] M. Bostock, "D3-voronoi," https://github.com/d3/d3-voronoi, 2015.

[15] T. C. Project, *CGAL User and Reference Manual*, 4th ed. CGAL Editorial Board, 2018.