Storing and Clustering Large Spatial Datasets Using Big Data Technologies[*]

Alejandro Cortiñas, Miguel R. Luaces, and Tirso Varela-Rodeiro

Universidade da Coruña
Laboratorio de Bases de Datos
A Coruña, Spain
{alejandro.cortinas, luaces, tirso.varela.rodeiro}@udc.es

Abstract. In this paper we present the architecture of a system to store, query and visualize on the web large datasets of geographic information. The architecture includes a component to simulate a large number of drivers that report their position on a regular basis, an ingestion component that is generic and can acommodate three different storage technologies, a query component that aggregates the results in order to reduce the query time and the data transfered, and a web-based map viewer. In addition, we define an evaluation methodology to be used to benchmark and compare different alternatives for some components of the system, and we validate the architecture with experiments using a dataset of 40 million locations of drivers.

Keywords: spatial big data, web-based GIS, software architectures

1 Motivation

Current technology makes the real-time collection of massive volumes of geographic information feasible. The computing power of current mobile devices is similar to the one of a desktop computer from the last decade. They can be used to measure different variables such as the geographic position using a GPS receiver, or the user activity using an accelerometer. *Mobile Workforce Management (MWM)* technologies will benefit especially from the information collected using mobile devices, and they are gaining attention because they are used by companies to manage and optimize the task scheduling of their workers and to improve the performance of their business processes [3]. Hence, Mobile Workforce Management is useful to detect patterns in the past activity of workers, or to predict trends that can improve the future scheduling. For example, Mobile Workforce Management can be used by a company to detect which tasks are

[*] This work has been funded by MINECO (PGE & FEDER) [TIN2016-78011-C4-1-R, TIN2016-77158-C4-3-R, TIN2015-69951-R]; CDTI and MINECO [Ref. IDI-20141259, Ref. ITC-20151305, Ref. ITC-20151247]; Xunta de Galicia [ED431C 2017/58].

costly for the company, or to ensure that it has the lowest number of active employees at any time of the day.

Datasets produced by mobile sensing and Mobile Workforce Management technologies are large and complex. As an example, consider a carsharing service like car2go. If each car produces a GPS position every 10 seconds (64 bytes considering a device id, a timestamp, three geographic coordinates, speed, bearing, and accuracy), it generates 552,900 bytes of data every day. Considering that car2go in Madrid has 500 vehicles [1], it would require over than 263 MB of storage each day. Larger systems, such as the taxi fleet (over 16,000 licenses in the region of Madrid), or the inclusion of additional sensor data (such as accelerometer data) would produce larger datasets.

Although the datasets are large, storage space is affordable and it does not seem impossible to store all this information, but querying and visualizing these datasets is a still difficult task. Current web-based GIS technology cannot deal with this problem. Data management technologies have been working during the last years to support horizontal scaling and distributed processing [7,2,8,5]. Therefore, storing and quering large geographic datasets can be achieved. However, middleware software such as map servers and visualization software such as javascript-based map viewers are not addapted to large datasets and distributed processing. Consider, for example the map server Geoserver [6]. It supports querying multiple data souces, but the support for MongoDB is quite recent and there is no support for other NoSQL technologies. Similarly, consider the map viewer Leaflet [4]. It supports different types of data layers but it includes plugins to aggregate large datasets to avoid cluttering the map display. However, the aggregation is performed on the client side, thus requiring large datasets to be transferred over the network and to be processed in the web browser. Hence, in order to support the visualization of large geographic datasets, middleware components and map viewers must support querying and aggregating geographic data using distributed processing systems.

In this paper, we present the architecture of a system to store, query and visualize on the web large datasets of geographic information (see Section 2). Given that selecting the most appropriate technology to support these usage scenarios is complicated, we have started to define an evaluation methodology to be used to benchmark and compare different alternatives for some components of the system (see Section 3). Therefore, the architecture includes a system to simulate the real load on one of these systems by simulating a large number of drivers that circulate through a road network and report their position to the server on a regular basis. In addition, the system also provides a solution to feed different storage subsystems in a simple way with the same set of data so that they can be tested under the same load conditions and evaluate their performance with the same queries. Finally, we also show preliminary results in which we use the proposed system to store, query and visualize 40 million locations of drivers in a relational spatial database management system (PostgreSQL + PostGIS), a NoSQL database management system (MongoDB) and a Big Data

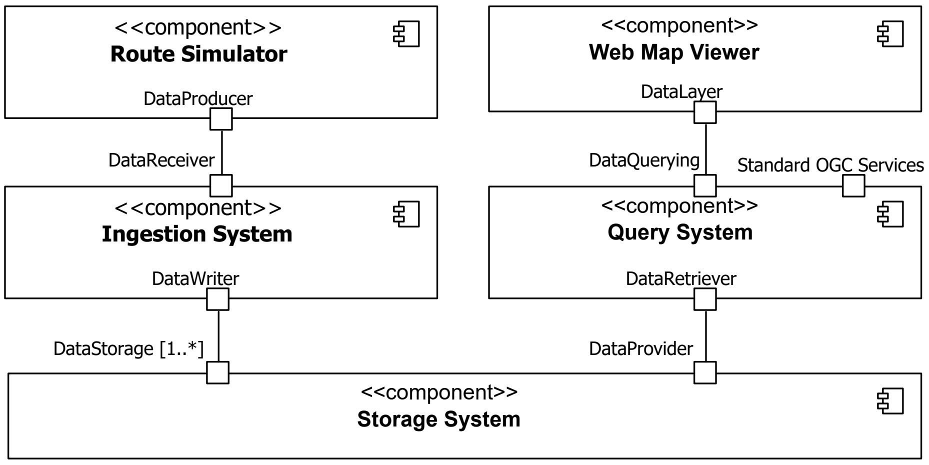


Fig. 1: System architecture

tool for storing and querying OLAP data (Druid) (see Section 4). Finally, we discuss the results and propose future work in Section 5).

2 Proposed Architecture

Figure 1 shows the main components of the system architecture. The `Route Simulator` component runs on the client side of the system and it simulates a collection of simultaneous drivers. Each driver starts at a random position in the road network, it calculates a route to a random destination, and it generates positions along the sections of the route with the specified periodicity assuming a random speed expressed as a percentage of the maximum allowed speed. For example, a driver can generate positions every second assuming that it always circulates at 80 % of the maximum speed of the road, while another driver can generate positions every five seconds running at 105 % of the maximum speed of the road. The component `Ingestion System` runs on the server side and it is responsible for collecting the data in real time. Its implementation is generic so that it is very easy to implement extensions to the component that behave differently simply by implementing a Java interface. The component `Storage System` is in charge of storing the data in a database management system.

Regarding the querying and visualization funcionationaly of the architecture, final users interact with the `Web Map Viewer` component, which is based on a standard web-based map viewer. Additionally, considering that the datasets that have to be visualized are extremely large, the information in these datasets is clustered depending on the zoom level in order to avoid cluttering the map display. The process of query solving and data clustering is performed on the server side by the `Web Map Viewer` component, which receives queries sent by the `Web`

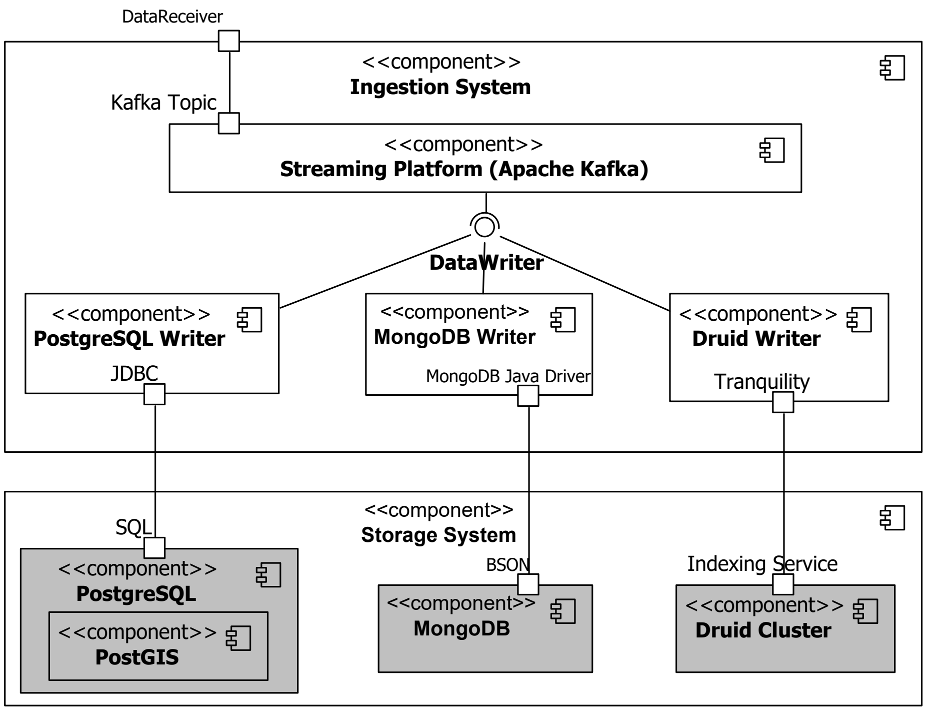


Fig. 2: Detailed ingestion architecture

`Map Viewer` component, delegates them on the `Storage System` component, and sends the results back to the client.

Figure 2 shows a detailed view of the ingestion architecture components. The components with a gray background are used without modifications. The entry point of the `Ingestion System` component is a messaging system (Apache Kafka) that is used to decouple the task of receiving large collections of data from the storage. The `Streaming Platform` component consumes the Kafka events and forwards them to different storage systems. Figure 3 shows an example of an event received by the `Ingestion System` component. It consists of information regarding the worker id, the GPS position of the worker, the timestamp of the position, and additional information in JSON format that is specific of the particular domain for which the architecture is being used.

The communication with the `Storage System` component is managed by a component that implements the generic `Data Writer` interface. We have currently implemented three alternatives: one that stores the events in Postgres + PostGIS (the component `PostgreSQL Writer`), another one that stores the data in MongoDB (the component `MongoDB Writer`), and another that stores the data in Druid (the component `Druid Writer`).

Given that the purpose of the data stored in the system is the interactive visualization by the user in a map viewer, and that having a fluid visualization

```
{
 "worker_id"; 3,
 "position": {
    "x": -8.0041161,
    "y": 43.18902,
    "z": 517,
    "speed": 32.48,
    "bearing": 83.6,
    "accuracy": 4.5509996
  },
  "timestamp": 1513763866,
  "data": {...additional data in json format...}
 }
```

Fig. 3: Example of an event received by the `Ingestion System` component

in the client side is a very important requirement, it is necessary to aggregate the points on the server side of the application and send to the client side only the result of the aggregation. That is, instaead of transferring large collections of individual geographic points, we want to transfer a smaller collection of clustered points amd the number of events that are added under that point. Furthermore, considering that the user will be able to define specific spatial and temporal ranges for the set of events that have to be retrieved to visualize by means of zoom and pan operations in a map and a time range control, precomputed clusters cannot be used because the variation among queries is too large. The simplest alternative is to perform the query *get all points in the range (xmin, ymin, tmin) - (xmax, ymax, tmax)* and apply an aggregation algorithm on the result, but it is a costly solution in terms of computation requirements. Instead, we propose to preprocess each event when it is inserted into the system and add versions of the geographic point with decreasing precision. For example, if each point is represented in a geographic coordinate system in which the coordinates represent longitude and latitude in degrees, a value with an accuracy of 9 decimals represents a maximum of 1 millimeter on the surface of the Earth. Storing 7 additional versions of the same geographical point with 7 different precisions (between 2 and 8 decimals) makes the process of aggregation as simple as grouping the events by equal values of coordinates and counting the number of elements. Therefore, the components implementing the `Data Writer` interface are also in charge of computing and storing the additional versions of the geographic point for each event.

Figure 4 shows a detailed view of the querying architecture components. The components with a gray background are used without modifications. The same design pattern that was used in the `Ingestion System` component is used to manage the communication with the `Storage System`. A generic interface was defined (`DataRetriever`) and components implementing this interface are responsible for parsing the query and communicating with the `Storage`

`System` component. We implemented three alternatives: Postgres + PostGIS (`PostgreSQL Retriever`), MongoDB (`MongoDB Retriever`), and Druid (`Druid Retriever`).

The data retrieving components are used in two alternative use cases. In the first use case, we support direct visualization of the data in the `Web Map Viewer` component using `Leaflet` and two components that manage the communication: a client-side component called `LeafletDataLayer` that implements the Layer interface of Leaflet, and a server-side component called `Leaflet Backend` that receives queries and delegates them to the appropriate data retrieving component. In the second use case, we implemented an extension to `Geoserver`, a popular map server component. This extension (named `Geoserver Backend`) acts as an adapter between Geoserver and the data retrieving components. Whereas the first scenario avoids middleware layers and performs faster, the second scenario provides applications with standard OGC services such as WMS, WFS and Filter Encoding. Figure 5[1] shows an example of a user interface using the Web Map Viewer component. The user interface displays a map with clusters that represent the amount of events that are stored in the database in a time range specified by the users.

3 Evaluation Methodology

In order to validate the architecture that we have proposed in Section 2, we have identified three key aspects that have to be evaluated and that lead to three research questions:

– *Research question 1 (RQ1). Which of the candidate storage technologies provides a faster answer to aggregation queries?* The selection of a storage technologies cannot be based only on how fast is able to answer a specific type of query because there may be many other requirements (e.g., transaction support, horizontal scaling, etc.). However, being able to answer the aggregation queries that are the basis of the architecture is a very important requirement with a high weight on the decision.
– *Research question 2 (RQ2). Can we achieve a constant time in aggregation queries using different versions of each geographic point?* In Section 2, we proposed a simple approach that can be used to improve aggregation queries using multiple versions of each geographic point. It is clear that this approch requires additional storage space, but it is not clear wether it will be possible to achieve a constant time answering queries.
– *Research question 3 (RQ3). Which query predicate performs faster, the distance based or the geometry based?* All storage systems provide two alternatives to solve range queries: one based on a distance predicate, which uses mathematical operations, and another one based on a bounding box predicate, which uses geometric operations. It is important to discover wich alternative of the queries performs faster in order to use it in queries.

[1] Note to the reviewers: we are currently working on the user interface, in case the paper is accepted we will upadte the screenshot with a better style for the clusters.

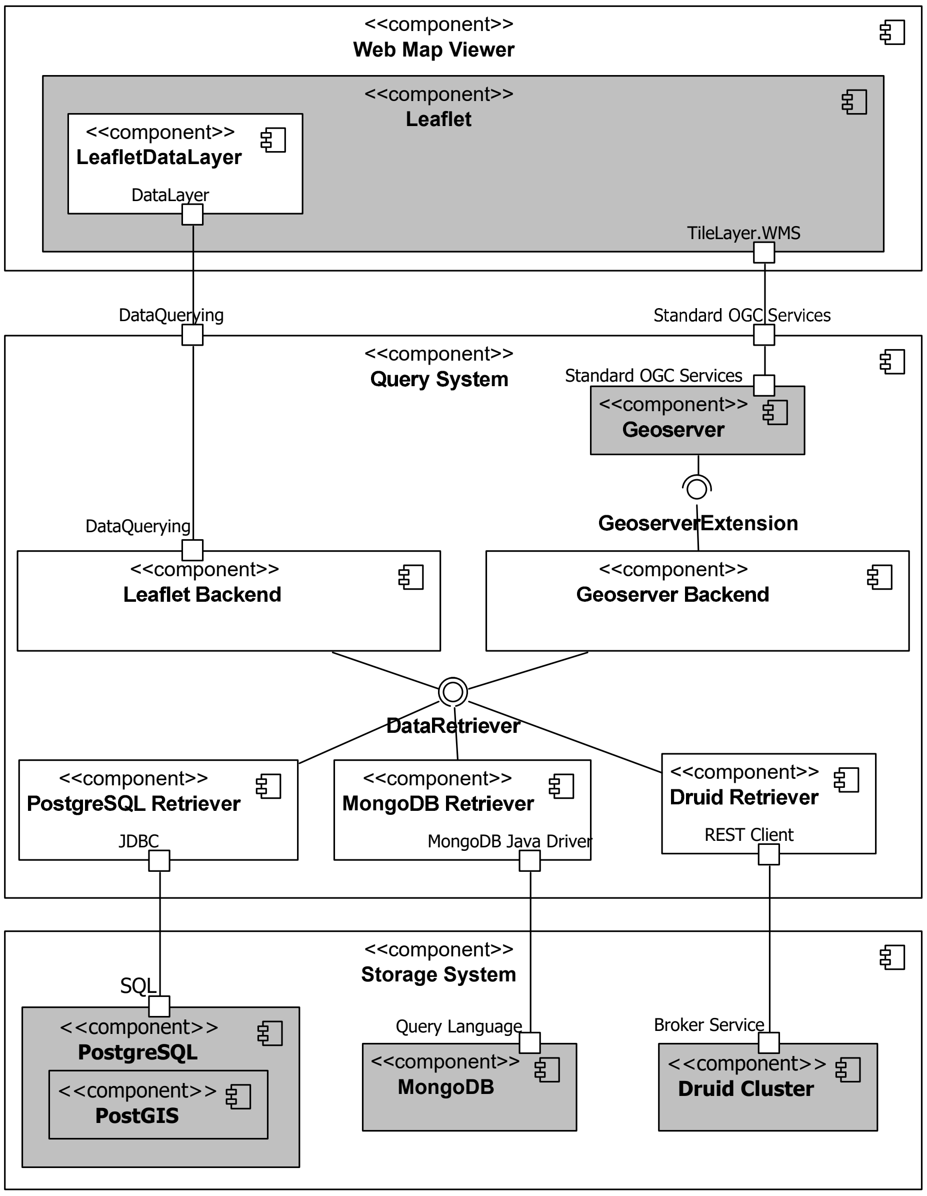


Fig. 4: Detailed querying architecture

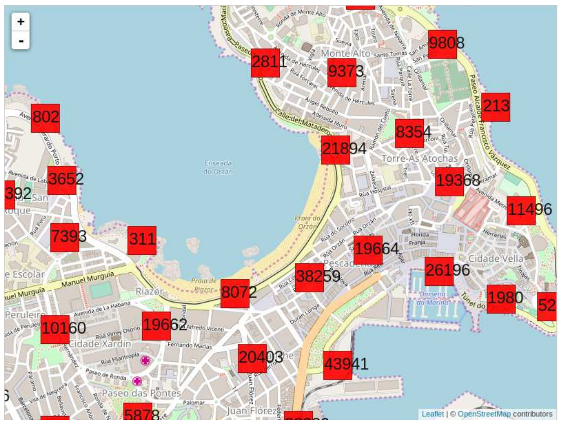
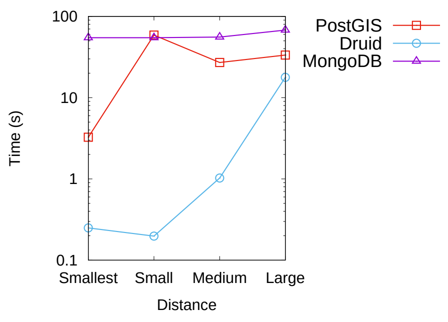


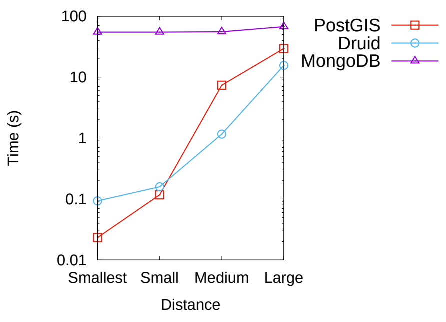
Fig. 5: Example of a user interface using the Web Map Viewer component

To generate test datasets, we have run the `Route Simulator` component in a desktop computer (Intel Core i7-3770, 4 cores, 3.40GHz, 8GB of RAM). The use of Kafka asynchronously as the entry point of the `Ingestion System` component makes the client very light and it can generate events for 1000 simultaneous drivers with hardly any load for the system. The server side was run on a machine that hosted all the server-side components of the architecture (`Ingestion System`, `Storage System`, and `Query System`). Only one storage technology was running simultaneously (either Postgres+PostGIS, MongoDB or Druid) in order to avoid resource allocation competitions. The system has behaved in a stable manner and has allowed to generate two datasets: one with 3.2 million points in 3 hours (*small dataset*), and another with 40.7 million points in 24 hours of continuous execution (*large dataset*).

In order to evaluate RQ1, for each candidate storage, we generated queries with four different spatial ranges (from small ones in the order of 0.0001 degrees to large one in the order of 0.1 degrees) using always the same version of the geographic point (the one with three decimals). One hundred queries were randomly generated and each query was executed exactly once to avoid the efects of any possible caching. To evaluate RQ2, the same strategy was used (one hundred queries with four different spatial ranges from small to large). However, this time the most suitable version of the geographic point was used for the aggregation according to the query range (e.g., if the query range is 0.0001 degrees the version with 4 decimals is used). Furthermore, the large dataset was used for these test. Finally, to evaluate RQ3 the queries ranges were generated using the strategy RQ1, but they were answered using both predicates. The queries were written to disk and executed twice independently in order to use the same query ranges with both predicates and to avoid any possible caching.



(a) Results for RQ1



(b) Results for RQ2

Fig. 6: Experiment results for RQ1 and RQ2

4 Experiments and Results

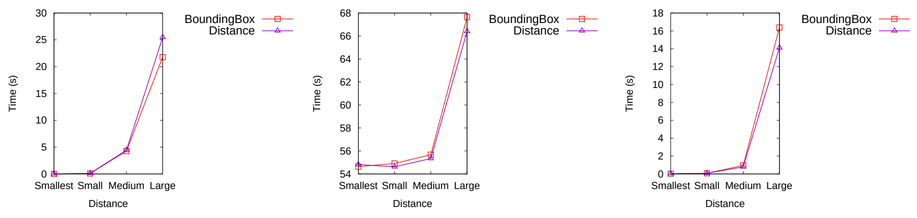
Figure 6a shows the results of the experiment performed to evaluate RQ1. The horizontal axis represents the different spatial ranges from small to large, and the vertical axis represents the average time in seconds to answer 100 queries using a logarithmic scale. It can be seen that Druid outperforms Postgres+PostGIS, which in turn outperforms MongoDB. The results indicate that Druid is the best option for the storage technology.

Figure 6b shows the results of the experiment performed to evaluate RQ2 using the same axis. The results are similar, even though the performance of Postgres+PostGIS improves yielding results comparable to Druid. However, the time required to answer queries increases with larger spatial ranges instead of remaining constant. The result indicate that our approach cannot be used to achieve a constant query time.

Figure 7 shows the results of the experiment performed to evaluate RQ3. The horizontal axis represents the different spatial ranges from small to large, and the vertical axis represents the average time in seconds to answer 100 queries using a different linear scale in each figure. It can be seen that the performance of both predicates (distance and bounding box) is similar. However, in Postgres+PostGIS the bounding box predicate outperforms the distance predicate, whereas in MongoDB and Druid the results are the opposite. The results indicate that the predicate used in the queries does not have a significative effect on the performance.

5 Conclusions and Future Work

We have presented in this paper the architecture of a system to store, query and visualize on the web large datasets of geographic information. We also have started to define an evaluation methodology to benchmark and compare different alternatives for some components of the system. We also show preliminary



(a) Postgres+PostGIS (b) MongoDB (c) Druid

Fig. 7: Experiment results for RQ3

results in which we use the proposed system to store, query and visualize 40 million locations of drivers in PostgreSQL+PostGIS, MongoDB, and Druid. We validated the system and we can conclude that Druid is the most promising technology. We also evaluated our approach to achieve a constant time in aggregation queries and we can concluded that it is not sucessful. We believe that the problem with the approach is that truncating a decimal means that one in a level of aggregation represents one hundred points in the next level of aggregation. Thus, the difference between the different levels of aggregation is too high. Finally, we evaluated wether the specific predicate used in the queries is relevant and we found out that there is no significative difference between them.

As future work, we are currently working on evaluating which technology scales better in a cluster, and new approaches to achieve constant query time in aggregation queries.

References

1. car2go Iberia S.L.: car2go Madrid website (2017), `https://www.car2go.com/ES/en/madrid/`, (Consulted on 29/12/2017)
2. Chodorow, K.: MongoDB: The Definitive Guide. O'Reilly Media, Inc. (2013)
3. Creelman, D.: Top Trends in Workforce Management: How Technology Provides Significant Value Managing Your People (2014), `http://www.oracle.com/us/products/applications/workforce-management-2706797.pdf`, (Consulted on 29/12/2017)
4. Crickard, P.: Leaflet.Js Essentials. Packt Publishing (2014)
5. Eldawy, A.: Spatialhadoop: Towards flexible and scalable spatial processing using mapreduce. In: Proceedings of the 2014 SIGMOD PhD Symposium. pp. 46–50. SIGMOD'14 PhD Symposium, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2602622.2602625`
6. Henderson, C.: Mastering GeoServer. Packt Publ., Birmingham (2014)
7. Obe, R.O., Hsu, L.S.: PostgreSQL: Up and Running A Practical Introduction to the Advanced Open Source Database. O'Reilly Media, Inc., 2nd edn. (2014)
8. Yang, F., Tschetter, E., Léauté, X., Ray, N., Merlino, G., Ganguli, D.: Druid: A real-time analytical data store. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. pp. 157–168. SIGMOD '14, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2588555.2595631`