A Scalable Data Pipeline for Realtime Geofencing Using Apache Pulsar

K. Sundar Rajan, A. Vishal, Chitra Babu

A Scalable Data Pipeline for Realtime Geofencing using Apache Pulsar

K. Sundar Rajan, A. Vishal, and Chitra Babu

Department of Computer Science and Engineering, Sri Sivasubramaniya Nadar College of Engineering, Tamil Nadu, India
sundarrajan16110@cse.ssn.edu.in, vishal16124@cse.ssn.edu.in, chitra@ssn.edu.in

Abstract. A geofence is a virtual perimeter for a real-world geographic area. Geofencing is a technique used to monitor a geographical area by dividing it into smaller subareas demarcated by geofences. It can be used to create triggers whenever a device moves across a geofence to provide useful location-based services. Since real-world objects tend to move continuously, it is essential to provide these services in real-time to be effective. Towards this objective, this paper introduces a scalable data pipeline for geofencing that can reliably handle and process data streams with high velocity using Apache Pulsar - an open-source Publish/Subscribe messaging system that has both stream processing and light-weight computational capabilities. Further, an implementation of the proposed data pipeline for a specific real-world case study is presented to demonstrate the envisaged advantages of the same.

Keywords: Geofencing · Apache Pulsar · Stream Processing · Scalable data pipeline.

1 Introduction

Data science as a field has evolved rapidly over the years to solve increasingly complex problems that facilitate improved living standards for society. In recent times, data is being progressively generated in tremendous volume, velocity and variety. Analyzing this vast data can provide valuable business insights, which can lead to effective decision-making. There is a significant requirement for adequate computational resources to find quick answers to real-time queries involving big data.

Usually, such queries have been executed using sequential scans over a large fraction of a database. In the context of big data, this approach takes a lot of computation time. Increasingly, several applications demand real-time response rates. One example could be updating ads based on recent trends observed on Twitter and Facebook.

Stream handling and processing in real-time is a necessity in a lot of real-world applications. Real-time geofencing is one such example. Geofencing is a technique used to monitor a geographical area by dividing it into smaller parts

demarcated by virtual boundaries known as geofences. The software which uses GPS, RFID, Wi-Fi, or cellular data can be used to trigger pre-programmed action such as - prompting mobile push notifications, triggering text messages or alerts, sending targeted ads on social media, allow tracking of fleets of vehicles, whenever a mobile device or RFID tag breaches a geofence set up around a geographical location.

Objects transmit their live location as a continuous data stream. This data needs to be processed sequentially and incrementally on a record-by-record basis. Message queues or publish/subscribe systems such as Apache Kafka[5], and RabbitMQ[15] are used to ingest the data stream in order to enable asynchronous communication as well as provide data persistence and fault-tolerance. This stream of location data should be transformed into a suitable format before analytics is performed to provide meaningful insights. Therefore, to perform such transformations in real-time, stream processing systems are such as Apache Spark[10] and Apache Flink[1] are used.

However, in certain use cases such as real-time geofencing, where a simple transformation from a specific geofenced area is required, such heavy computation frameworks incur excessive overhead. Therefore, Apache Pulsar[16] - an open-source Publish/Subscribe messaging system that has both stream processing and light-weight computational capabilities - is appropriate for use as a single entity that handles both ingestion and transformation of data. Furthermore, the advantages of using a single system to ingest and process the stream are - lower administrative overhead, easier maintenance and lower cost.

The remainder of this paper is organized as folows:
- Section 2 discusses related works in real-time data processing and geofencing.
- Section 3 describes the key features and advantages of using Apache Pulsar.
- Section 4 details the architecture of the system and its various components.
- Section 5 discusses a Case Study of a particular real-time geofencing system and the design considerations made while choosing each of its components.
- Section 6 concludes and provides future directions.

2 Related Work

Wang et al.[8] have proposed a scalable geofencing based system using agricultural machine data to send real-time alerts to farmers. They have proposed a scalable processing system using Apache Kafka to ingest the live stream of data and a cloud Apache Storm architecture that ingests this data and performs the necessary transformation. Since this work has used two separate components for ingestion and processing of data, it takes more time and resources for initial set-up and subsequent maintenance of the pipeline. Google has built a Geolocation Telemetry system[11] to add location-based context to telemetry data using Google Pub/Sub. This is done to ingest the data and python scripts that reverse-geocodes the data to convert latitude and longitude to a street address, calculates the elevation above sea level, and computes the local time offset from

UTC by using the timezone for each location. This design is easy and effective for simple stream operations but will not be scalable when the size, velocity and heterogeneity of the stream increases.

Bogdan et al.[7] proposed a geofencing service based on the Complex Event Processing(CEP) paradigm and discussed its applications in the Ambient Assisted Living(AAL) space. The CEP paradigm is very useful when performing intensive stateful transformations by involving pattern detection and generation of secondary pattern streams from the existing ones. However, for simple geofence transformations, such complex pattern detection is not required and hence a dedicated stream processing system which supports CEP is entirely unwarranted. WangJun et al.[9] have proposed a distributed data stream processing pipeline using Apache Flume, Apache Kafka and Apache Spark. In this work, Apache Flume[17] is used as a point-point queue which collects data from multiple sources, Apache Kafka is used for segregating data under different topics and real-time streaming while Apache Spark is used for data transformation and analysis. Nevertheless, when the transformations performed are simple, all these three components can be replaced with a single entity such as Apache Pulsar which supports Data Queuing, Streaming and Simple Processing thus tremendously reducing the overall complexity and maintenance cost of the data pipeline.

3 Apache Pulsar

Apache Pulsar is an open-source distributed pub-sub messaging system which has both stream processing and lightweight computational capabilities. Pulsar comprises a set of brokers, bookies and an inbuilt Apache ZooKeeper[3] for configuration and management. The bookies are from Apache Bookkeeper[4] which provide storage for the messages until they are consumed.

Unlike current messaging systems that have taken the approach of co-locating data processing and data storage on the same cluster nodes or instances, Pulsar takes a cloud-friendly approach by separating the serving and storage layers. Pulsar has a layered architecture with data served by stateless “broker” nodes, while data storage is handled by “bookie” nodes. The architecture of Pulsar is shown in Fig. 1.

3.1 Key Advantages

Low latency with durability Pulsar is designed to have low publish latency (<5ms) at scale. To confirm this claim, benchmark tests[12] have been performed in literature using the performance analysis tools provided by OpenMessaging Benchmark[14]. These tests were performed with different load sizes and partitions to analyze the latency of Pulsar. The results of the test are shown in Table 1.

The same test was performed for Apache Kafka to compare its latency against that of Pulsar. The results of this test are tabulated in Table 2.

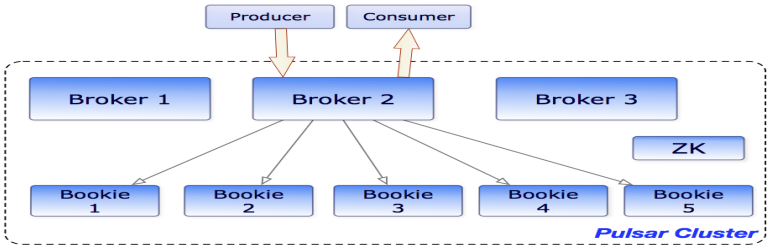

Fig. 1. Pulsar Architecture[13]

Table 1. Pulsar Latency Benchmark with 1 topic and 16 partitions[14]

Latency Type	Avg	50th	75th	99th	99.99th
Publishing (milliseconds)	2.677	2.655	3.23	3.825	20.883
End-to-end (milliseconds)	3.165	3.0	3.0	12.0	190.0

Hence, for latency-sensitive workloads, Pulsar performs better than Kafka. It can provide low latency, as well as strong durability guarantees. This is essential to a system that involves real-time processing and querying, and hence Pulsar outperforms other message queues in these aspects.

Persistent Messaging Pulsar provides guaranteed message delivery for applications. If a message successfully reaches a Pulsar broker, it will be delivered to its intended target. The non-acknowledged messages are stored durably until they are delivered to and acknowledged by consumers. Pulsar also provides automatic retries until the consumer consumes the messages.

Pulsar Functions Pulsar Functions are light-weight compute logic that can be easily deployed along with Pulsar to perform real-time data transformation on the stream, thereby eliminating the need for a dedicated stream processing engine. This proves to be very critical for the use case discussed in this paper because it eliminates the necessity to have a separate stream processing logic/system to transform the location coordinates to specific geofenced areas. A Pulsar function can be deployed along with the Pulsar cluster to perform this function efficiently and reliably.

Table 2. Kafka Latency Benchmark with 1 topic and 16 partitions[14]

Latency Type	Avg	50th	75th	99th	99.99th
Publishing (milliseconds)	8.479	8.152	9.64	169.57	211.64
End-to-end (milliseconds)	11.03	10.0	12.0	28.0	259.0

Horizontal Scalability Pulsar has a layered architecture that enables scaling brokers and bookies independently while maintaining the Zookeeper cluster to coordinate the system. This makes it very easy to horizontally scale the system when the workload increases. This is an essential feature that is required when more devices are tracked, and the message rate increases.

Fault Tolerance Unlike traditional pub/sub systems, pulsar uses a distributed ledger. Pulsar breaks down the huge logs into several smaller segments, and it distributes those segments across multiple servers while writing the data using Apache Bookkeeper as its storage layer. This makes it easier to add a new server in case of failures, as there is no need to copy the logs of the failed server. Further, since logging is performed in bookkeeper, it reduces the brokers' stress to log data while handling high-velocity real-time data streams, resulting in improved overall performance.

Shared subscriptions and Partitioned Topics Pulsar subscriptions have the provision to add as many consumers as needed on a topic, with Pulsar keeping track of all of them. If the consuming application cannot keep up, a shared subscription can be used to distribute the load among multiple consumers. Also, partitioned topics that multiple brokers handle can be used to increase the throughput, thus making it highly scalable.

4 System Architecture

This section illustrates the proposed system for stream handling and processing. The data pipeline for this system is based on streaming architecture. The data pipeline comprises the following modules: Stream handling, Stream Processing, Storage, and Querying. The overall system diagram is shown in Fig. 2.

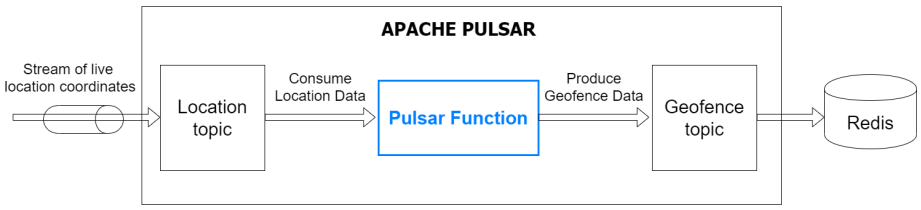

Fig. 2. System Diagram

4.1 Stream Handling

Moving devices transmit their current location coordinates, taken as a streaming input to the pipeline using a publish/subscribe system - Apache Pulsar. The

location coordinates are continuously published onto a pulsar topic - a named channel used to pass messages published by producers to consumers who process those messages. Once the data is in a topic, the data is persistently stored (by Apache Bookkeeper) until it is processed.

4.2 Stream Processing

The location coordinates present in the topic have to be processed reliably, wherein all the records should be guaranteed to be processed exactly once. The messages are to be processed in the order they are published to maintain the correct context. Since the stream of data is published into the topic continuously, the processing also has to be performed in real-time. Pulsar Functions are capable of performing such transformations with the guarantees mentioned above. Pulsar functions can receive messages from one or more input topics, and every time a message is published to the topic, the function code is executed. For this, a Pulsar function has been written, which reads a message from the location topic, maps the coordinate to its corresponding geofenced area from the list of geofenced areas that the user has provided. Subsequently, the Pulsar Function publishes the transformed geofenced information into a geofence topic. Conversion of the location data to the geofenced area can be realized using Point in Polygon algorithms with the help of R-Trees[2] or Quad Trees to navigate quickly through the search space. Isolating the processing logic to Pulsar Functions makes it easier to adopt the proposed system by easily porting the existing geofencing logic into a corresponding Pulsar Function. Any change to the geofencing logic can be realized by changing the Pulsar Function without affecting any other part of the pipeline. This enhances the portability and pluggability of the overall system.

4.3 Storage and Querying

Data in the geofence topic is updated on a record by record basis in a swift lookup table implemented with Redis[18] or Memcached[19], to find metrics such as the number of devices in a geofenced area, the transition from one geofenced area to another for an individual device and so on, in real-time.

Apart from this, data persistence can be achieved by capturing periodic snapshots of the lookup table and storing it in a persistent database/data warehouse. This can also be realized alternately by using Pulsar IO Connectors to export the data into other databases or messaging systems such as Apache Cassandra[6], Aerospike[20], Apache Kafka, and RabbitMQ, where further transformations can be performed before appropriately storing the data. This data accumulated over a long period can be used for analytics and business intelligence operations to improve decision-making.

In order to present a proof of concept for the proposed pipeline, the following case study has been implemented as a prototype.

5 Case Study - Finding cab hotspots using real-time geofencing

Cab-hailing applications such as Ola or Uber typically use proximity information to assign the nearest cab to every incoming customer request. The lack of restrictions on unoccupied cabs' movement may result in an uneven distribution of cabs across geographical regions and failure to satisfy customer requests. To make optimal use of available cabs, live cab location and customer request data need to be analyzed and cabs need to be redistributed across the various regions based on the identification of hotspots.

The proposed data pipeline has been implemented for this case study application primarily using Apache Pulsar. Other appropriate technologies, such as Redis datastore and Google BigQuery[21] data-warehousing solution have also been deployed to efficiently query and maintain data.

5.1 Design Overview

Cabs belonging to a particular fleet continuously transmit their current location using a hardware device present in them. This stream of cab locations is published into a location topic in Apache Pulsar. Using Pulsar Functions, the published location data is read from the location topic, transformed into geofence data using the point-in-polygon algorithm combined with R-Trees, and subsequently published back into a different Pulsar topic, namely, geofence topic. The geofence data from that topic is consumed by a subscriber and stored in a read-write optimized lookup table. Periodic snapshots are taken from the lookup table and sent to a data warehouse for performing further analytics. The system generates alerts if the number of cabs in a particular geofenced area exceeds a pre-specified threshold. Also, the number of cabs in a specific geofenced area can be queried from the Redis table in real-time. The system diagram is shown in Fig. 3.

5.2 Data Ingestion and Transformation

The cab location data is published continuously in the Pulsar location topic. The location data published is later consumed by Pulsar Functions. The set of geofenced areas under consideration has been used to build an R-Tree. A combination of point-in-polygon algorithm and R-Trees has been used to transform a given location coordinate to its corresponding geofenced area number. This combination helps in quickly narrowing the search space and finding the corresponding geofenced area. This transformation logic has been written into a Pulsar Function that takes the location topic as the input and publishes the transformed geofence data into a geofence topic in Pulsar. Thus, the Pulsar function consumes every message published in the location topic, applies the transformation function, and publishes the geofence data in the geofence topic.

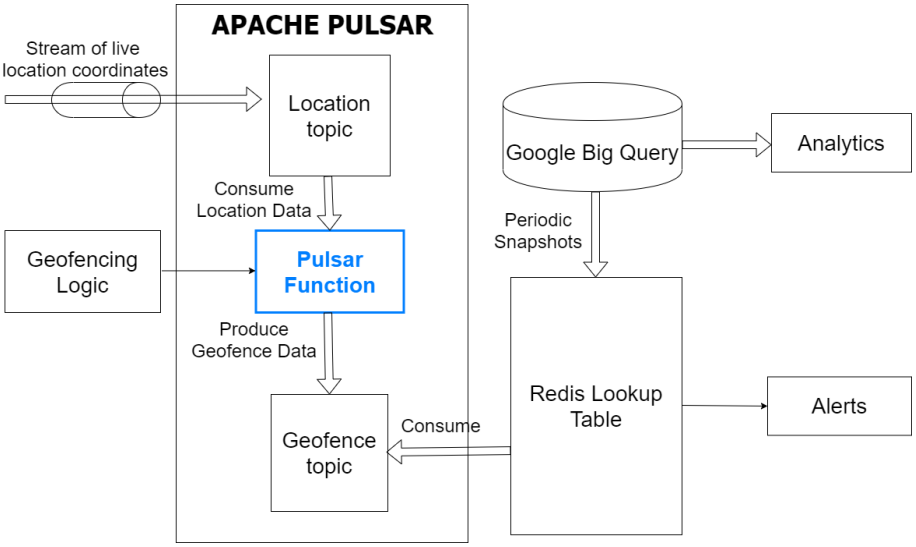

Fig. 3. Cab Hotspot Detection System

5.3 Hotspot Detection

The geofence data published by the Pulsar Functions are consumed by the hotspot detection module, which updates a Redis Table. Whenever geofence data is read from the Pulsar topic, a Redis table corresponding to the number of cabs in the current geofence is updated. This table contains the number of cabs in each geofenced area. Whenever a geofenced area with the number of cabs exceeding the threshold is encountered, an alert message is sent indicating a cab hotspot in that area.

In addition to Redis, Memcached, and MongoDB[22] have also been also evaluated as potential alternatives. However, Redis is more suitable as it is better than Memcached in terms of memory management and handling of free space and has faster disc access time than MongoDB, which is a crucial consideration while designing a real-time system.

5.4 Persistent Storage

Since Redis has volatile memory, snapshots of the Redis table(s) are taken periodically and stored in a data warehouse to have persistent storage. Google BigQuery has been chosen as the data warehouse as it can handle large quantities of data and is easy to set up. As this case study has been carried out for an educational purpose, free availability without any associated cost was an important consideration.

Amazon Redshift[23], LucidDB[24] and PostgreSQL[25] have also been considered as alternatives. Google BigQuery is preferred over these choices as Redshift does not have a free pricing tier, LucidDB was discontinued in 2014, and

PostgreSQL is only capable of handling data of the order of a few terabytes. However, Redshift is one of the best cloud data warehousing options when there are no financial constraints.

5.5 Validation of the data pipeline

To validate the working of the pipeline, an Azure pulsar cluster across three nodes running Ubuntu 18.04 with 20GB SSD storage each was set up. The nodes have Intel Xeon CPU E5-2673 and primary memory of 16GiB each. Java runtime environment was installed on each node.

A stream of data was generated and ingested into the pub/sub system at various rates. Since no real-time streaming source was available, the stream data had to be generated from a static dataset. Chicago cabs dataset[26] was chosen for this purpose. A python message producer program with the pulsar client and producer object was created. The cabs were initialized to random locations in the state of Chicago. Their current locations were stored in the Redis database. Random destination points for each cab were picked. The cabs were made to move from their current location point to a point P calculated as $\frac{currentlatitude+destinationlatitude}{2}$, $\frac{currentlongitude+destinationlongitude}{2}$. The point P and the destination point were fed into a "Haversine Function" which determines the great-circle distance between two points on a sphere given their longitudes and latitudes. If the haversine distance is above a certain threshold, the process is repeated. Otherwise, it indicates that point the P is very close to the destination, and it is approximated to be the destination. This point becomes the new current location, and a new destination point is chosen as described earlier, and the process is repeated continuously. This generates a continuous stream of cab movement data, which was ingested into Pulsar in real-time. The movement of the cabs has been visualized using a Google Maps API where markers are plotted dynamically. The position of cabs at an instant is shown in Fig. 4.

5.6 Results and Discussions

The performance of the proposed data pipeline has been investigated using a single producer with 2 threads and a message size of 64 bytes to measure the publish latency for different message rates. The results are shown in Table 3.

It can be observed that the change in latency is very negligible, even with a significant increase in the message rate. This proves the robustness and the scalability of the proposed system.

The percentile latency is plotted in Fig. 5 and it is used to monitor the spike in latency under heavy load.

From Fig. 5, it can be seen that the publish latency is less than 1 second for 99.9% of the messages demonstrating the system's ability to handle heavy load while maintaining its performance guarantees. The spike observed in the remaining 0.1% can be reduced either by increasing the broker systems' processing power or by adding additional brokers to the cluster.

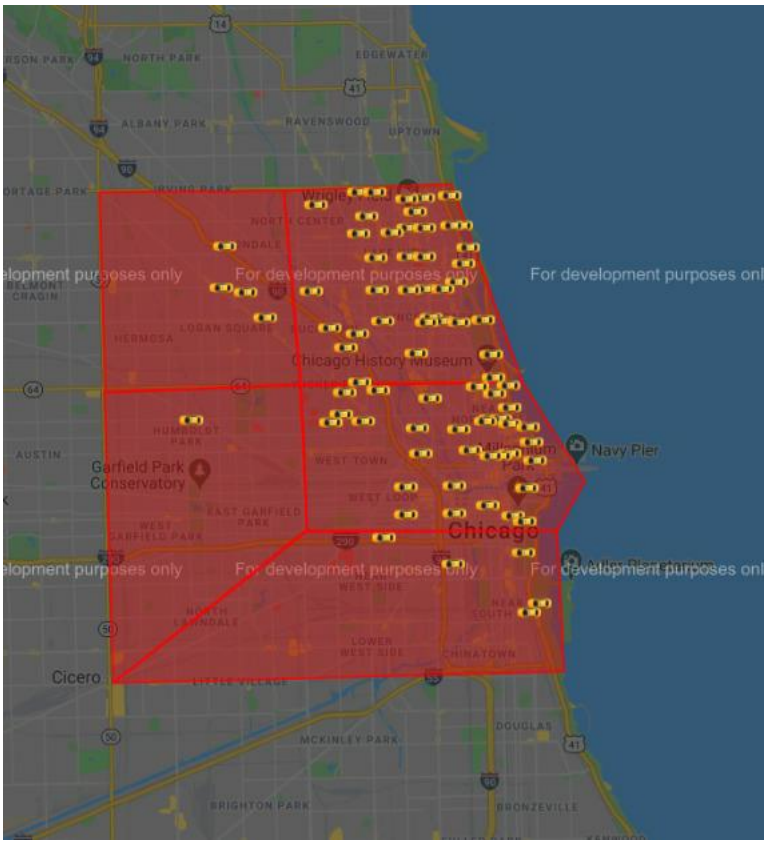

Fig. 4. Cab Visualization

Table 3. Message Rate vs Avg.Latency(Message size: 64 bytes)

Message Rate(msg/sec)	**Avg. Latency(ms)**
10	12.126
100	10.796
1000	26.218
10000	30.285
100000	25.551

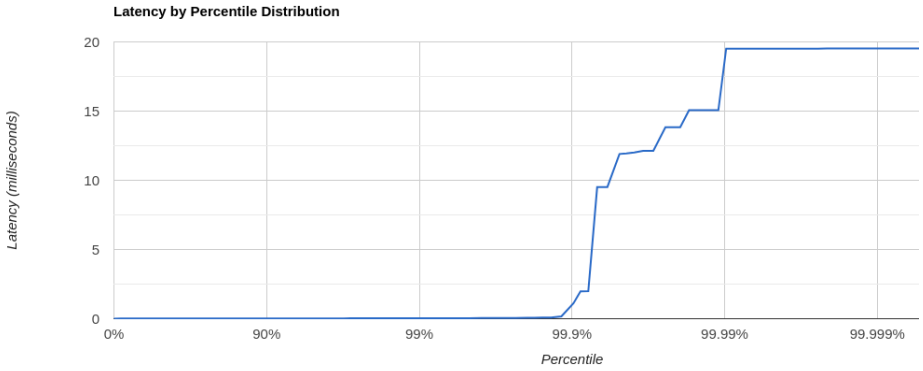

Fig. 5. Latency by Percentile Distribution (100000 msg/sec)

Experiments have also been conducted to verify the fault tolerance and message delivery guarantees of the system. Whenever a broker node failed, all the unacknowledged messages were replayed from the bookkeeper once the broker node was back up. All the published messages were consumed without any loss of messages. When running in the once guaranteed processing mode, all the messages published were processed exactly once by the pulsar functions. This shows the capability of the system to recover from failures while still maintaining the context and the processing guarantees.

6 Conclusions

A scalable data pipeline for real-time geofencing using Apache Pulsar has been proposed in this paper. The proposed data pipeline has been designed to handle a high velocity of requests and provide real-time responses. The proposed design is simple and involves fewer components making it easy to set up and maintain. The isolation of the geofencing logic to Pulsar Functions makes it easier to adopt this system with any existing geofencing logic. Usage of Pulsar as the sole stream handling and processing engine helps handle streaming requests with very high velocity and reliably process all the messages. Pulsar is highly fault-tolerant because of its 3-Tier architecture and distributed logging using Apache Bookkeeper, making the system scalable and robust without compromising real-time responsiveness.

The case study performed corroborates the claims made. It contains useful insights into various design considerations when choosing other systems to work in combination with the proposed data pipeline. Thus, it could serve as a reference for future works involving the proposed data pipeline.

References

1. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, pp. 28-38, 2015.
2. Guttman, A.: R-Trees. A Dynamic Index Structure For Spatial Searching, ACM SIGMOD Record Volume 14, No.2, pp. 1-11, June-1984. https://doi.org/10.1145/971697.602266
3. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: ZooKeeper: Wait-free Coordination for Internet-Scale Systems, Proc. Usenix Annual Technical Conference, June 2010.
4. Junqueira, F.P., Kelly, I., Reed, B.: Durability with BookKeeper, ACM SIGOPS Operating Systems Review, pp. 9-15, 2013.
5. Kreps, J., Narkhede, N., Rao, J.: Kafka: a Distributed Messaging System for Log Processing, Proc. NetDB, pp. 1-7, 2011.
6. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system, ACM SIGOPS Operating Systems Review, vol. 44, no. 2, pp. 35-40, 2010.
7. Târnaucă, B., Puiu, D., Nechifor, S., Comnac, V.: Using Complex Event Processing for implementing a Geofencing Service, IEEE 11th International Symposium on Intelligent Systems and Informatics, pp. 1–6. Subotica, Serbia, September 2013.
8. Wang, Y., Balmos, A.D., Layton, A.W., Noel, S., Ault, A., Krogmeier, J.V., Buckmaster, D.R.: An Open-Source Infrastructure for Real-Time Automatic Agricultural Machine Data Processing, 2017 ASABE Annual International Meeting, pp. 1–13. Spokane, Washington, July 2017. https://doi.org/10.13031/aim.201701022
9. Wang, J., Wang, W., Chen, R.: Distributed Data Streams Processing Based on Flume/Kafka/Spark, 2015 3rd International Conference on Mechatronics and Industrial Informatics, pp. 948–952. Zhuhai, China, October 2015. https://doi.org/10.2991/icmii-15.2015.167
10. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets, Proc. 2nd USENIX conference on Hot topics in cloud computing, HotCloud'10, pp. 10-10, Berkeley, CA, USA, 2010.
11. Building a Scalable Geolocation Telemetry System using the Maps API, https://cloud.google.com/solutions/scalable-geolocation-telemetry-system-using-maps-api. Last accessed 9 August 2020.
12. Kafkaesque, https://kafkaesque.io/performance-comparison-between-apache-pulsar-and-kafka-latency/. Last accessed 4 January 2021
13. Pulsar Architecture, https://pulsar.apache.org/docs/en/concepts-architecture-overview/. Last accessed 4 January 2021
14. OpenMessaging Benchmark, http://openmessaging.cloud/. Last accessed 4 January 2021
15. RabbitMQ, https://www.rabbitmq.com/documentation.html. Last accessed 4 January 2021
16. Apache Pulsar, https://pulsar.apache.org/. Last accessed 4 January 2021
17. Apache Flume, https://flume.apache.org/. Last accessed 4 January 2021
18. Redis, https://redis.io/. Last accessed 4 January 2021
19. Memcached, https://memcached.org/. Last accessed 4 January 2021
20. Aerospike, https://www.aerospike.com/. Last accessed 4 January 2021
21. Google BigQuery, https://cloud.google.com/bigquery. Last accessed 4 January 2021
22. MongoDB, https://www.mongodb.com/. Last accessed 4 January 2021

23. Amazon Redshift, https://aws.amazon.com/redshift/. Last accessed 4 January 2021
24. LucidDB, https://dbdb.io/db/luciddb. Last accessed 4 January 2021
25. PostgreSQL, https://www.postgresql.org/. Last accessed 4 January 2021
26. Chicago Taxi Trips, https://www.kaggle.com/chicago/chicago-taxi-trips-bq/. Last accessed 4 January 2021