Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 151 (2019) 463–470

Procedia Computer Science

www.elsevier.com/locate/procedia

The 10th International Conference on Ambient Systems, Networks and Technologies (ANT)
April 29 - May 2, 2019, Leuven, Belgium

A Reactive System for Big Trajectory Data Management

Soufiane Maguerra[a,*], Azedine Boulmakoul[a], Lamia Karim[b], Hassan Badir[c], Ahmed Lbath[d]

[a] *LIM/IOS, FSTM, Hassan II University of Casablanca, Mohammedia, Morocco*
[b] *Higher School of Technology EST Berrechid, Hassan 1st University, Morocco*
[c] *National School of Applied Sciences Tangier, Abdelmalek Essaâdi University, Morocco*
[d] *University Grenoble Alpes, CNRS, LIG/MRIM France*

Abstract

Nowadays, distributed systems have become requisite to process and analyse the large amount of generated data. In particular, spatio-temporal data has known an exponential growth in the last years. This could be explained by the proliferation of indoor and outdoor tracking devices and the value of the knowledge that can be extracted from their analysis. The data describe a moving objects' behavior in space and time. Once these coordinates are assembled they form a raw trajectory. The knowledge extracted by analysing the set of trajectories is very valuable; they can be even mapped to other contextual data to add value. Preprocessing is an essential step in the mining process to filter unnecessary records and clean the noise. Furthermore, reactive systems offer the possibility to process massive data in a non-blocking, resilient, responsive and elastic manner. Despite this interest, to the best of our knowledge no works have been conducted into offering a reactive system to preprocess massive trajectories. The aim of this study is to fill this gap by proposing a reactive system based on distributed actors to manage big trajectory data. Our system is deployed with the Play Framework, Akka, MongoDB, AngularJS and D3.js. Initially, the system can load batches of trajectories stored in HDFS in a distributed manner. The scope of our study is to provide an overview of the system, to study the impact of the increase in computing resources over the scalability, and to provide the optimal node configuration. The results indicate a higher scalability of the system, and the evaluation is conducted by considering the Geolife project's GPS trajectory dataset.

Keywords: Akka; Big Data; distributed computing; reactive systems; message-oriented architecture; Actor Model; concurrent processing; parrallel computing

* Corresponding author. Tel.: +212 23 31 53 52
E-mail address: maguerra.soufiane@gmail.com

1877-0509
10.1016/j.procs.2019.04.063

1. Introduction

At the present time, a massive load of spatial events is being generated instantly by location-aware devices. The knowledge inferred by mining these events can have a great value in making significant decisions. To be able to extract the knowledge efficiently, Big Data technologies must be leveraged because the conventional technologies fail to support the massive, real-time nature of the data. reactive systems can come as an answer to these requirements.

A reactive system has four main traits : *responsive*, *resilient*, *elastic* and *message-driven*. In detail, the system is always able to respond to external demand, always has a plan to resolve failures, it can scale up or down depending on the load, and it is based on asynchronous message passing. The reactivity implies that the system is non-blocking by nature, and it isolates behavior, state, and failures of its components. The distribution and thread safe state of its components make it possible for the system to handle massive batches and streams with the lowest possible latency. Furthermore, Akka[1] gives the tools to build such systems with actors as an abstraction for the components. The actor model is specially designed to surpass the Object Oriented Model by providing the means to achieve a higher level of concurrent and parallel computing [7]. Moreover, the lightweight nature of the actors can enable a single node system, once accordingly orchestrated, to deploy millions of them. Despite the significant value, research on the field of processing trajectories in the Big Data context are limited to transformative or imperative systems such as Spark or Hadoop.

In this paper, we tackle this issue by providing a reactive system to manage massive trajectories. To the best of our knowledge, this is the only study that details the use of an actor-based reactive system to manage big trajectory data. The system is deployed with Akka, Play Framework[2], MongoDB[3], Nginx[4], D3.js[5], and AngularJS[6]. All the technologies are reative and support massive data, even MongoDB is queried via the Reactive Mongo driver[7]. Play is used as a back-end behind an Akka Http server and serves as a client for ana Akka Cluster. The Cluster provides high computational resources, and is based upon the push model within a multi master and worker architecture. The scope of this paper is limited. Due to the limited number of pages, we limit our scope into overviewing the message-oriented architecture of our reactive system, to evaluate its scalability, and study the computation power of the different nodes. The trajectories studied in these paper are raw by nature, and they have been loaded into HDFS from the Geolife project's GPS trajectory dataset [22, 20, 21].

The rest of the paper is divided into: Section 2 discusses the related work; Section 3 details reactive systems, the Actor Model, and trajectories; Section 4 presents the architecture of our reactive system; Section 5 provides the obtained results; Section 6 serves as a conclusion for our study.

2. Related Work

The work on Big Trajectory Data Management Systems (DMS) are related to research proposing the extension of a conventional algorithm into the distributed environment and applying it in the context of processing massive trajectories. The distributed environment is often related to Spark or Hadoop. da Silva et al. [14] have implemented the DBScan algorithm in Hadoop, and they applied it for the detection of dense areas in traffic data. Xie et al. [17] took interest into deploying the trajectory similarity search in Spark by using Simba, and they used two-level layer index to effciently query the stored data.

In the area of distrbuted trajectory offline processing, Sinnott et al. [16] proposed a cloud based system including Spark, Geomesa, GeoServer, and HDFS. Their system conducts density analysis of traffic data and visualizes the results in a heat map. Zhang et al. [19] presented TrajSpark, a scalable system featuring in-memory query processing. The system leverages Spark and applies Range Queries (RQ) and k-Nearest Neighbors (kNN) queries over trajecto-

[1] https://akka.io/
[2] https://www.playframework.com/
[3] https://www.mongodb.com/
[4] https://www.nginx.com/
[5] https://d3js.org/
[6] https://angular.io/
[7] http://reactivemongo.org/

ries. The queries are optimized by a global three-level layer index. Shang et al. [13] proposed a system to answer similarity search and join queries with Spark SQL. Optimization is achieved by a two-level layer index. Ding et al. [3] exploited Spark to provide a unified platform UITraMan for big trajectory ETL, storage, data management and analytics. Moreover, their system employes Chronical Map as a mean to leverage the off-heap memory in the processing. In [4], the same authors proposed VIPTRA a system that uses UITraMan to process the data in an interactive manner. Ruan et al. [12] used Spark and Azure Table to preprocess trajectories. Algorithms for noise filtering, segmentation, map matching and index building have been integrated into the distributed environment to support massive loads.

The literature also contains some cloud-based systems. Xie. et al [18] proposed a system based on a peer-to-peer network in Open Stack to answer RQ and kNN queries over big trajectories. Liu et al. [9] conducted a real-world study aiming to select billboard locations based on taxi trajectory data. Their system uses D3.js, version 1 of AngularJS, and leaflet for the visualization. The server is Node.Js and they store the data in MongoDB.

Other research are more interested in the real-time processing aspect. Ma et al. [10] proposed a mobile cloud system for taxi ridesharing. Their system support data in the scale of a city by leveraging an index server, a communication server, and a set of scheduling servers. Da Silva et al. [15] proposed an incremental algorithm for clustering a stream of trajectories. In the distributed environment, li et al. [8] used Azure Table, Azure Storm and Azure Redis to process trajectories. Their pipeline includes preprocessing, index and query processing. Gong et al. [5] extended the work of [16] to analyse trajectories in near real time. Their system SMASH is based on docker containers, and it includes Spark, Geomesa, GeoServer, and HDFS. In their study, they applied DBScan to cluster traffic data. Another study was conducted by He et al. [6], aiming to detect illegal parking places of vehicles. Their system includes MongoDB, and Storm to online process the trajectories of bikers.

Decidedly, all the systems that have been stated above are based on either Spark, Hadoop, Storm, or other cloud based technologies. The literature lacks a reactive non-blocking system based on actors that features asynchronous message passing, resiliency, and responsiveness.

3. Background

Reactive Systems. The Reactive Manifesto [1] defines a reactive system by four traits *responsive*, *resilient*, *elastic* and *message-driven*. The most discriminant trait is the message-driven one, it signifies that the system is based on asynchronous message passing. In particular, The system isolates the lifecycle, state, behavior, and failures of its components. Furthermore, the system isolates the components in terms of space. This feature is denoted by the term *Location Transparency*, and it implies that the components are identified by a logical identifier not a physical one, which can be changed by the runtime environment to optimize the performance.

The Actor Model in Akka. Actors in Akka are components that communicate via messages, which contain the actor path *ActorRef* of both the sender and recipient. An actor may have several behaviors, has a thread safe state, and processeses messages from its predefined mailbox sequentially. The role of an actor can be summarized in reading, processing, and sending messages. Moreover, an actor can create other actors and supervise them by a specific supervision strategy to handle failures. Depending on this strategy, the parent can either resume the state, restart, shutdown, or escalate the exception to another parent.

Trajectories. Various representation of trajectories exist in the literature [2]. In our work, we take interest in raw trajectories. A raw trajectory is an ordered sequence of spatio-temporal events. Each event can be formulated as $e_i = (lon_0, lat_0, t_0)$, with $long_i$, lat_i, t_i respectively identifying the latitude, longitude, and time.

4. The design of the reactive system

Our reactive system can be sighted in Fig. 1. The deployment diagram shows three main parts a front-end, a back-end and an Akka Cluster. Note that the two last parts are implemented in the functional language Scala. Hence, the system assures from the ground up scalability and immutability as well as easily conducted parallel and concurrent processing. The front uses the version 6 of AngularJS for a reactive UI, and to compile the typescript it leverages a NodeJS server. The back is deployed with the Play Framework behind an Akka Http Server. The Play Framework

provides the means to implement a reactive web based application. Between the back and the front, there is an Nginx server that can be leveraged as a reverse proxy to resolve the Cross-Origin Resource Sharing (Cors) issue and to balance the load between a set of back-end servers. The back has the role of a cluster client that ships computationally massive requests to the Akka Cluster. The cluster is composed of five types of nodes a seed, router, reader, master and a worker. The set of seed nodes is indispensable because a seed enables the other nodes to join the cluster. The router has the main aim to route the requests into the other nodes depending on the type of the message. The reader nodes have actors that perform queries in an asynchronous manner on the MongoDB cluster. In particular, the Reactive Mongo driver is used for the querying tasks. The master node has a set of JobReceptionist actors that receive jobs and create for each job a master actor. At last, the worker node executes the tasks shipped by the master.

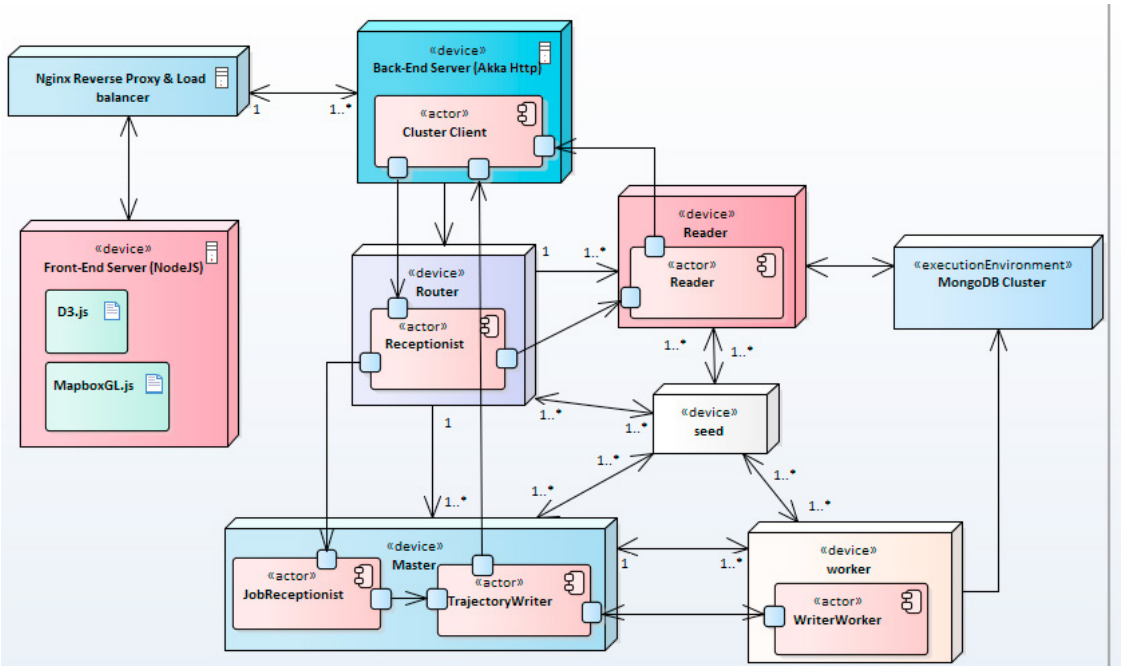

Fig. 1. The deployment diagram of the reactive system.

The communication between the different components can be seen in Fig. 2. The router node has a Receptionist actor that has two router actors a pool and a group. The pool creates routees in the reader nodes, and the group is linked to existing JobReceptionist actors. If the message is a ReaderMessage, then it is shipped via the pool router. Otherwise, it is sent via the group router. The JobReceptionist actor creates a TrajectoryWriter per job, and this one has a pool router that creates WriterWorker actors on the worker nodes. Therefore, the system is based on the multi master worker push model because the TrajectoryWriter pushes the load to WriterWorker routees. Note that the TrajectoryWriter and Reader actors can communicate to the cluster client directly because of the forward functionality. In detail, when a message is forwarded it keeps the original address of the sender.

At the current time, our system load trajectory data from hdfs. Then, it persists the data into MongoDB and possibly finds the stored trajectories and streams them back to the cluster client. The back-end also provides the possibility to stream the results to the front-end in a non-blocking manner. Furthermore, An instance of the TrajectoryWriter actor processes and distributes the provided HDFS paths in parallel. The WriterWorker actors, differently than the sequential processing behavior, process the received loads concurrently. In a related work of ours [11], we found out that the routing logics that support this asynchronous processing are RoundRobinPool, RandomPool, and AdaptiveLoadBalancingPool. Moreover, the one that accurately balances the loads is the RoundRobinPool. Decidedly, we use this one to distribute the messages among the WriterWorker routees. In case the processing of a message yields an exception, the routee creates an instance of the RepeatedMessage class and ships it to parent, which ships it back via the self contained pool router. The Fig. 3 has more details on the different messages.

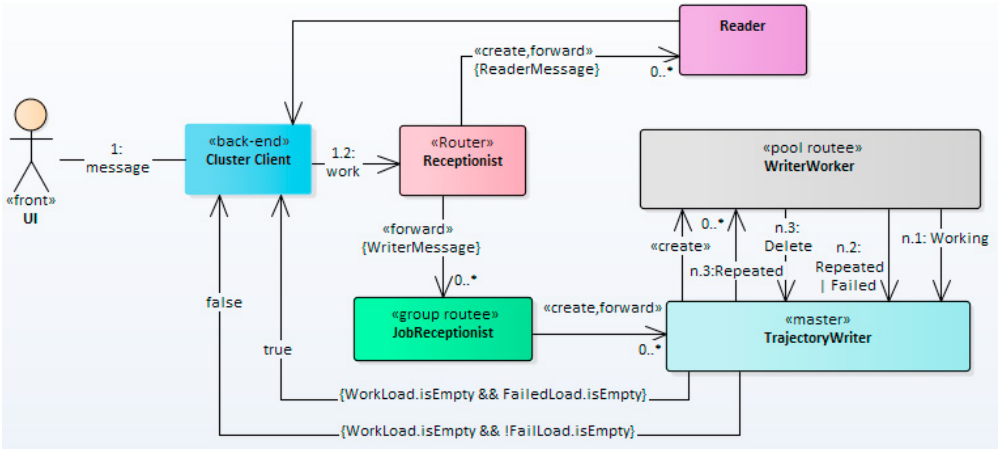

Fig. 2. The communication diagram of the reactive system.

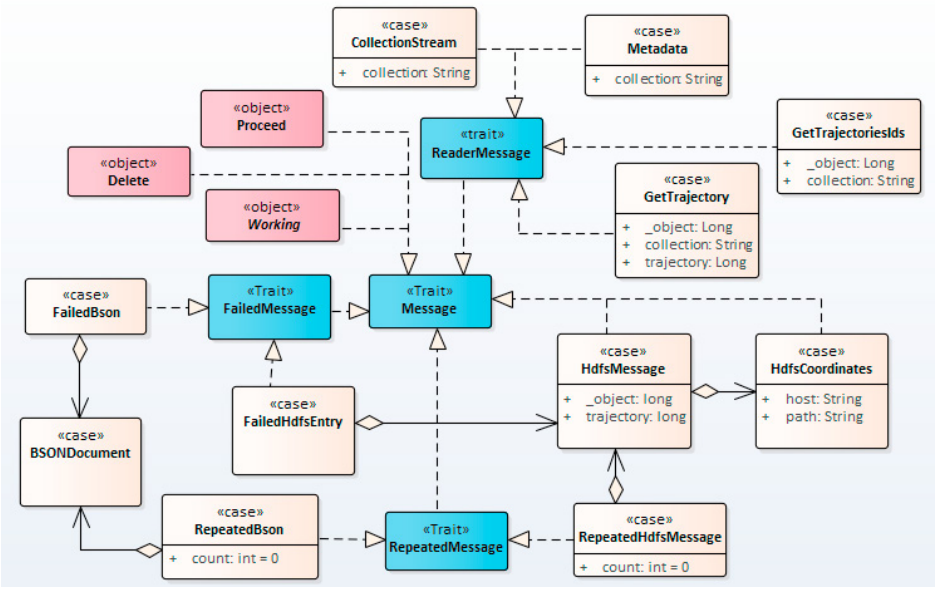

Fig. 3. The class diagram of the different messages.

Scala provides as with *traits* that contain behavior. In our system the highest level trait is Message. Another trait can possibly extend a trait, in our case we have ReaderMessage, FailedMessage, and RepeatedMessage traits. Moreover, Scala features *objects* can be considered as a static class in Java. In our system, each WriterWorker routee sends a Working message object to the TrajectoryWriter indicating that it is working. This way we have provided the system with a design to handle failures. In detail, the TrajectoryWriter has a ReceivedTimeout. If it receives no messages from the workers, it changes its behavior and starts looking for any worker nodes in the cluster periodically. If a worker node has successfully joined the cluster, the master resumes the job by keeping a state of the remaining messages that need to be executed, and it changes its behavior to the standard by sending a Proceed object message to itself. At last, the response of a write request fails or succeeds depending on the size of the remaining workload and failed load. In particular, after the worker achieves the processing of a message it possibly sends a message that has failed and needs to be repeated to the parent. The RepeatedMessages have a counter that gets incremented for each execution.

The maximum number of repeats is predefined and once it gets surpassed the message becomes a FailedMessage and is no longer executed. Then, the worker sends a Delete message to the parent, which subtracts it from the remaining load. At the reception of each Delete message, the TrajectoryWriter checks the number of the remaining workload then the failed load to mark the end of the request. At the end of the request, the parent destroys itself, which implies the destruction of everything supervised by him.

5. Results & discussions

In our experiments, we first tried to study the impact of changing the CPU number on each of a single master node and worker node. The Fig. 4 proves the scalability of the master node. Since the TrajectoryWriter processes the received paths in parallel, the change in the number of CPUs affects the execution time considerably. The same result for the worker node can be concluded from Fig. 5. The execution time decreases by the increase in the number of cores.

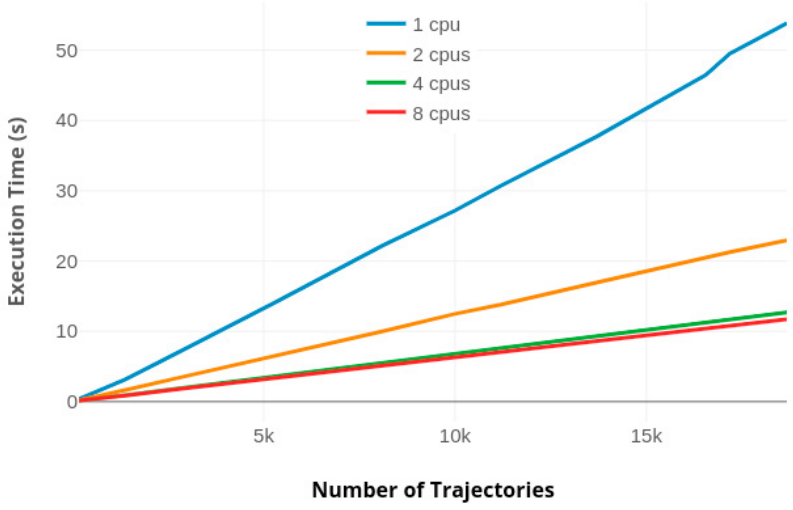

Fig. 4. Variation of the CPUs number in the master node and its impact on the processing time.

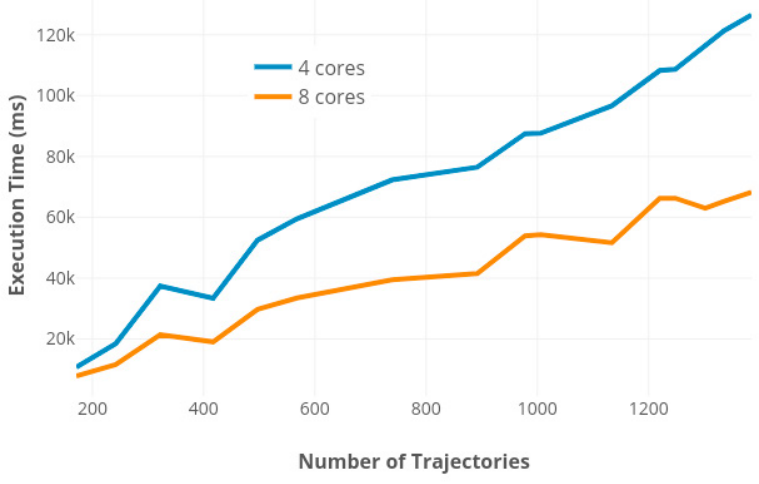

Fig. 5. The relation between the processing time of the trajectories and the number of CPUs in the worker node.

After proving the scalability of our nodes, we wanted to retrieve the optimal number of routees in a worker node, which is characterized by 4 CPUs and 4 GB of memory. the results of the Fig. 6 indicate that the optimal number is 2 routees per worker. At last, we prove in the Fig. 7 that the increase in the number of worker nodes can greatly impact the reduction of the execution time.

Fig. 6. Changing the number of routees in a worker node and its effect on the processing time.

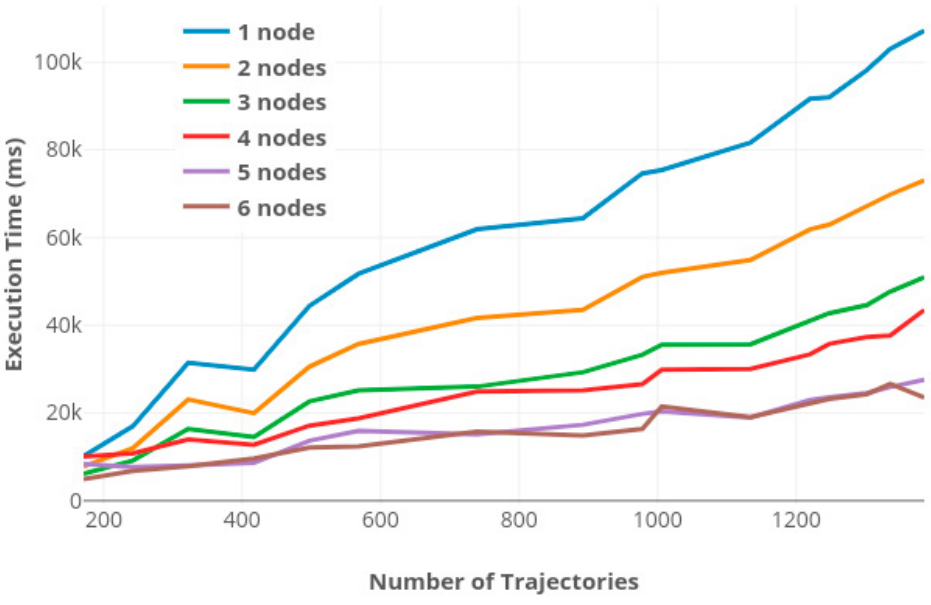

Fig. 7. The scalability of the processing with the increase in the number of worker nodes.

6. Conclusions

Our study has presented a reactive system that manages big trajectory data. The system has been deployed with non-blocking code in mind. Hence, it features responsiveness, resiliency, asynchronous message passing, and elasticity. The experiments that have been achieved prove the scalability of our cluster nodes, and the efficiency of our design. Currently, we are evaluating the preprocessing part of the trajectory data mining pipeline in our system. Aiming, to propose a fully non-blocking, asynchronous, message-driven system to process trajectories.

References

[1] Bonér, J., Farley, D., Kuhn, R., Thompson, M., 2014. The reactive manifesto, v2. 0.
[2] Boulmakoul, A., Karim, L., Lbath, A., 2012. Moving object trajectories meta-model and spatio-temporal queries. arXiv preprint arXiv:1205.1796 .
[3] Ding, X., Chen, L., Gao, Y., Jensen, C.S., Bao, H., 2018a. Ultraman: a unified platform for big trajectory data management and analytics. Proceedings of the VLDB Endowment 11, 787–799.
[4] Ding, X., Chen, R., Chen, L., Gao, Y., Jensen, C.S., 2018b. Viptra: Visualization and interactive processing on big trajectory data, in: 2018 19th IEEE International Conference on Mobile Data Management (MDM), IEEE. pp. 290–291.
[5] Gong, Y., Rimba, P., Sinnott, R., 2017. A big data architecture for near real-time traffic analytics, in: Companion Proceedings of the10th International Conference on Utility and Cloud Computing, ACM. pp. 157–162.
[6] He, T., Bao, J., Li, R., Ruan, S., Li, Y., Tian, C., Zheng, Y., 2018. Detecting vehicle illegal parking events using sharing bikes' trajectories .
[7] Hewitt, C., Bishop, P., Steiger, R., 1973. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence, in: Advance Papers of the Conference, Stanford Research Institute. p. 235.
[8] Li, R., Ruan, S., Bao, J., Zheng, Y., 2017. A cloud-based trajectory data management system, in: Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, ACM. p. 96.
[9] Liu, D., Weng, D., Li, Y., Bao, J., Zheng, Y., Qu, H., Wu, Y., 2017. Smartadp: Visual analytics of large-scale taxi trajectories for selecting billboard locations. IEEE transactions on visualization and computer graphics 23, 1–10.
[10] Ma, S., Zheng, Y., Wolfson, O., et al., 2015. Real-time city-scale taxi ridesharing. IEEE Trans. Knowl. Data Eng. 27, 1782–1795.
[11] Maguerra, S., Boulmakoul, A., Badir, H., 2019. Load balancing of distributed actors in an asynchronous message processing boundary. Submitted.
[12] Ruan, S., Li, R., Bao, J., He, T., Zheng, Y., 2018. Cloudtp: A cloud-based flexible trajectory preprocessing framework. ICDE. IEEE .
[13] Shang, Z., Li, G., Bao, Z., 2018. Dita: Distributed in-memory trajectory analytics, in: Proceedings of the 2018 International Conference on Management of Data, ACM. pp. 725–740.
[14] da Silva, T.L.C., Neto, A.C.A., Magalhães, R.P., de Farias, V.A., de Macêdo, J.A.F., Machado, J.C., 2014. Efficient and distributed dbscan algorithm using mapreduce to detect density areas on traffic data., in: ICEIS (1), pp. 52–59.
[15] da Silva, T.L.C., Zeitouni, K., de Macêdo, J.A., 2016. Online clustering of trajectory data stream, in: Mobile Data Management (MDM), 2016 17th IEEE International Conference on, IEEE. pp. 112–121.
[16] Sinnott, R.O., Morandini, L., Wu, S., 2015. Smash: A cloud-based architecture for big data processing and visualization of traffic data, in: Data Science and Data Intensive Systems (DSDIS), 2015 IEEE International Conference on, IEEE. pp. 53–60.
[17] Xie, D., Li, F., Phillips, J.M., 2017. Distributed trajectory similarity search. Proceedings of the VLDB Endowment 10, 1478–1489.
[18] Xie, X., Mei, B., Chen, J., Du, X., Jensen, C.S., 2016. Elite: an elastic infrastructure for big spatiotemporal trajectories. The VLDB Journal—The International Journal on Very Large Data Bases 25, 473–493.
[19] Zhang, Z., Jin, C., Mao, J., Yang, X., Zhou, A., 2017. Trajspark: a scalable and efficient in-memory management system for big trajectory data, in: Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data, Springer. pp. 11–26.
[20] Zheng, Y., Li, Q., Chen, Y., Xie, X., Ma, W.Y., 2008. Understanding mobility based on gps data, in: Ubicomp 2008, Ubicomp 2008. URL: https://www.microsoft.com/en-us/research/publication/understanding-mobility-based-on-gps-data/.
[21] Zheng, Y., Xie, X., Ma, W.Y., 2010. Geolife: A collaborative social networking service among user, location and trajectory. .
[22] Zheng, Y., Zhang, L., Xie, X., Ma, W.Y., 2009. Mining interesting locations and travel sequences from gps trajectories, in: WWW 2009, WWW 2009. URL: https://www.microsoft.com/en-us/research/publication/mining-interesting-locations-and-travel-sequences-from-gps-trajectories/.