GPSInsights: Towards an efficient framework for storing and mining massive real-time vehicle location data

Hoang-Linh Truong
Hanoi university of science and technology
linhth.k55.hust@gmail.com

Duy-Khanh Bui
Hanoi university of science and technology
duykhanh1412@gmail.com

Viet-Trung Tran
Hanoi university of science and technology
trungtv@soict.hust.edu.vn

ABSTRACT

Intelligent Transport System (ITS) has seen growing interest in collecting vehicle location data in order to build up real-time traffic monitoring and analytic systems. However handling these data creates challenges, as they are massive in volume and arriving in near real-time. In this paper, we proposed GPSInsights, a distributed system that is scalable and efficient in processing huge volume of location data stream. GPSInsights is built up on open-source, scalable and distributed components. We demonstrated our system with a scalable map matching implementation and performed experiments with real big datasets.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Bigdata

Keywords

GPSInsights, spatio-temporal data storage, distributed data processing, map matching

1. INTRODUCTION

With the widespread adoption of GPS technology, Intelligent Transport System (ITS) has seen growing interest in collecting location data of transport vehicles. This data collection is done with the purpose of being able to deliver not only real-time traffic monitoring but also useful traffic statistics and predictive information. Lee et al. [11] presented a data mining algorithm to discover traffic bottlenecks. Demiryurek et al. [4] proposed an online computation for optimal traffic routes based on traffic data. However, none of these approaches discuss how to implement the system at large-scale.

According to Decree No. 91/2009/ND-CP of Ministry of Transport of Vietnam, all Vietnamese-licensed cars must be equipped with a standardised global positioning system (GPS) which reports geo-location, speed and direction every 30 seconds to a centralised data centre. With nearly 200.000 cars in circulation in the near future, the data is enormous and has big data characteristics. First, data is generated continuously in big volume (e.g. petabytes) from hundred thousand of vehicles. Second, in-coming data rate is near real-time, at which the underlying system must deliver. Third, data has big value for the potential insights about the current situation of the traffic infrastructure as well as for the predictions.

As big data create non-conventional challenges, current ITS management systems storing data in relational database systems (e.g. via PostGIS [16]) will not be able to adapt to the data ingestion rate, nor being able to be processed efficiently. In this work, we describe GPSInsights: a novel scalable system for storing and mining massive real-time vehicle location data. GPSInsights is able to handle increasingly huge volume of data while supporting real-time analytics. We demonstrate GPSInsights with a scalable map matching implementation.

This paper is organised in 7 parts. In Section 2, we discuss an overall architecture of our system framework. In Section 3, we go into the details of the technologies and the components of GPSInsights. In Section 4, we establish a simple demonstration map matching algorithm. Section 5 presents experimental results with real datasets. Section 6 discusses related works on existed map matching algorithms and on storing spatio-time data. We conclude and discuss future work in Section 7.

2. SYSTEM DESIGN

We design GPSInsights with the following components as depicted in Figure 1

- Distributed input message queue: As the location data arrive from a huge number of source vehicles continuously, this component is responsible for combining the multi-source data, and putting them in their chronological order. It has to store and replicate the large input data dispersedly on a large cluster for high throughput, and for fault tolerance. The distributed message queue implements producer-consumer skeleton in which GPS devices installed on transportation vehicles are the producers and the storage engine or the processing engine of GPSInsights become the consumers.

- Streaming data processing engine: To allow realtime analytics, GPSInsights powers a streaming data processing engine. This component has to be able to analyse data "on the fly", which then outputs analytic reports such as average speed, number of vehicles, and traffic bottleneck prediction. It has to be scaled out on thousand of servers to adapt to the workload.

- Distributed result queue: Results from the data processing engine are sent to this component. This acts as the interface to continuous consumer services at application level such as web, mobile apps.

- Distributed spatio-temporal database: As location data consist of geolocation informations and timestamps in which the data are recorded, GPSInsights stores data in a distributed spatio-temporal database. This components aims to provide the spatial querying and data manipulation as PostGIS but at large-scale for offline phase.

- Distributed analytic result database. This database is selected to store the analytic results from the data processing engine, acting as the datastore for both display services at the application level and offline processing systems.

•

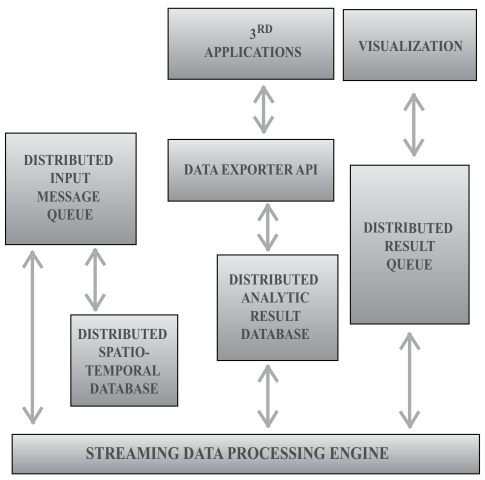
3RD APPLICATIONS
VISUALIZATION
DISTRIBUTED INPUT MESSAGE QUEUE
DATA EXPORTER API
DISTRIBUTED RESULT QUEUE
DISTRIBUTED ANALYTIC RESULT DATABASE
DISTRIBUTED SPATIO-TEMPORAL DATABASE
STREAMING DATA PROCESSING ENGINE

Figure 1: The GPSInsights system architecture

3. IMPLEMENTATION

With the system architecture shown in Chapter 2, we build our GPSInsights using open-source components with custom plugins to satisfy our design goals. By leveraging existing components, we can develop GPSInsights quickly and focus on the scalability aspect of the entire system. Thus, a part of our contribution is to design the system overall, to carefully extend the right components, and to run the experiments with real datasets.

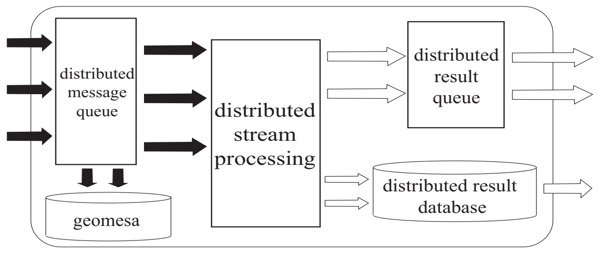
distributed message queue
geomesa
distributed stream processing
distributed result queue
distributed result database

Figure 2: GPSInsights implementation

3.0.1 Apache Kafka

Apache Kafka [10, 2] is a distributed publish-subscribe messaging service with the purpose of being fast, scalable and durable. Kafka maintains feeds of messages in categories called topics. Each topic is a partitioned log, distributed over multiple cluster server nodes as in Figure 1. Inside a particular partition, the messages are immutable and ordered, identified by an "offset". This design choice allows not only storing the amount of the data larger than the capability of any single node but also parallelizing read and write operations. Technically, producers that publish messages can distribute messages over multiple partitions of a topic in a round-roubin fashion, or they can distribute according to some semantic partition functions.

Kafka achieves fault-tolerance by replicating partitions across a configurable number of nodes. Each partition has one "leader" and several "follower" nodes. The "leader" node serves all read and write requests for the partition while the followers are mirroring. Once the leader failed, one of the followers will be automatically promoted to be the new leader (using the well-known distributed leader election algorithm). In production, every Kafka node acts as the leader for some partitions and the follower for others to achieve balancing.

In GPSInsights, Kafka is used for the distributed input message queue and the result queue. With suitable configuration, Kafka helps GPSInsights aggregating in-coming location data and routing analytic results to the appropriate layers (E.g. to the storages and to the end-user applications).

3.0.2 Spark Streaming and Storm

Spark Streaming [9, 19] and Apache Storm [17, 3] are the most popular open-source frameworks for distributed stream processing. In the distributed mode, both of them use a master/slave architecture with one central coordinator and many distributed workers. However, there are still the important differences in their architectures as following.

Spark Streaming lays on top of Apache Spark [24] for acting on data stream. At the core of Spark Streaming is the concept of discretized abstraction (D-streams) [25, 26], that considers in-coming records as a series of deterministic batches on small time intervals. Each batch is treated as a resllient distributed dataset (RDD) of Spark, and being processed using RDD operations.

Spark's RDDs offer fault-tolerance and parallel computation at large-scale though three important design principles. First, RDD is partitioned in chunks, distributed across compute nodes (as in Hadoop Mapreduce paradigm). Second, every computation within RDD is recorded in logs (called lineage). Third, the temporary data are kept in memory to speed up computation. If any partition of an RDD is lost due to a node failure, as long as the source of the input data which is usually at immutable Hadoop file system HDFS [18], then that partition can be re-computed from it using the lineage of operations.

Instead of batching up events that arrive within a short time and then process them as in Spark Streaming, Storm processes incoming events one at a time. Thus, storm processing latency could be sub-second, while Spark Streaming latency is of several seconds. The work in Storm is delegated to different types of components that are responsible for a specific processing task. The input data stream is received by a specialized component called a "spout". Then the spout immediately passes the data to another component called a "bolt". In the bolt, the data will be transformed in some way, and the bolt either sends them to some sort of storage or passes them to some other bolts. In general, the Storm cluster can be considered as a network of bolt components in which each one applies some kind of transformation on the data receiving from the spout, the arrangement of spouts and bolts and their connections in the cluster is called a topology. In storm, each individual record has to be tracked when moving through the system. However, Storm only guarantees that each record will be processed at least once.

GPSInsights is implemented to work with both Spark Streaming and Storm as the stream processing engine.

3.0.3 MongoDb

Because of the rapid increase in velocity and volume of the result data, Traditional relational databases are no longer the "one-size-fits-all" for every type of data. They do not scale well to large datasets because their scaling model is vertical: more data means bigger server. One way to scale relational databases across multiple servers is to do "database sharding". However, this mechanism is limited scalability due to the inherent complexity of the relational interface and the ACID (atomicity, consistency, isolation, and durability) guarantees.

MongoDB [12] is a document store with the possibility to scale horizontally. It is designed for managing semi-structured data organized as a collection of documents. In MongoDB, the structure of the documents is very flexible. There is no pre-defined scheme as the columns, and column datatypes as in relational databases. MongoDB distributes documents by the document IDs across servers and implements replication for fault-tolerance. When comparing the performance between the two different databases [23], MongoDb saw the much better performance than MySQL - the traditional relational database.

GPSInsights uses MongoDB to store the statistic results from the data processing engine. GPSInsights leverages the ability of MongoDB to write data fast and in a flexible scheme.

3.0.4 Geomesa

Geomesa [5] is an open-source, distributed, spatio-temporal data-base built on top of a column-family oriented distributed database called Accumulo [1]. Geomesa uses a very flexible strategy to linearize geo-time data. It distributes the data across the nodes in the cluster to leverage parallelism, thus enables efficient storing, querying, and transforming large spatio-temporal data. Geomesa is like PostGIS [16] but at very large-scale and for big data workloads.

GPSInsights relies on Geomesa for storing raw in-coming location data which will be the input for doing batch processing if neccessary. Note that GPSInsights focuses on real-time analytics but it also features long running analytic jobs. Those will be discussed in the future papers.

3.0.5 Guarantee reliability

The primary reason of choosing the distributed message queue as a component of GPSInsights is to minimize the number of data loss when the system fails. In the case of lacking the message queue component, the data processing engine would directly receive the data from the GPS devices. Once the master node of the processing engine dies (the master node of Spark called "driver program", one of Storm called "Spout"), GPSInsights will not receive and handle any data which arrive, thus those transportation data will be totally lost. By contrast, with the message queue, GPSInsight can ensure zero transportation data loss (the message queue is fault-tolerance as Kafka). When recovering, the processing engine will pull the next unprocessed data from the queue to restart.

However, GPSInsights still has to deal with unreliability even with the support of message queues. There are two challenges.

First, how to guarantee that the output data from Spark Streaming were completely sent to MongoDb? Data lost happens when Spark Streaming goes wrong and not pushes all the result data to the result database. MongoDb in this case receives the incomplete set of the data, but Spark Streaming supposes it completed the task with the current batch and then continued handling the next batch (Figure 3).

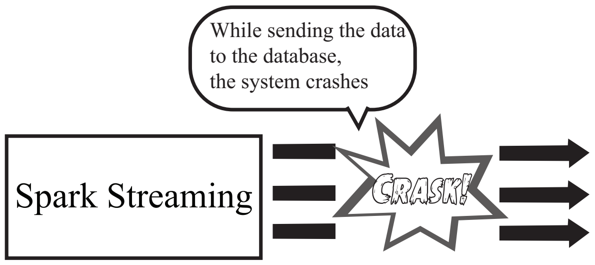

Figure 3: Spark Streaming failed to push data

Second, some messages might appear repetitively in a batch due to failures that the Spark Streaming Receiver failed to inform Kafka about its current received messages (Figure 4). Therefore, Spark Streaming supposed it received the data, but Kafka supposed that the messages was not sent successfully and would resend repetitively.

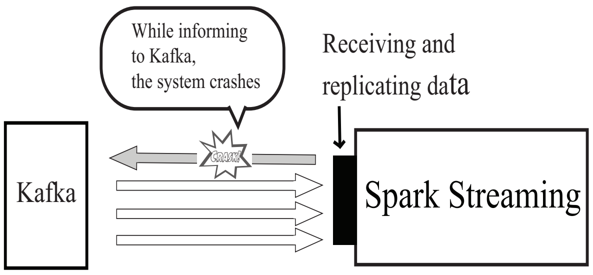
While informing to Kafka, the system crashes
Receiving and replicating data
CRASH
Kafka
Spark Streaming

Figure 4: Spark Streaming Receiver failed to inform Kafka message queue

The two above problems is caused by the fact that each component of GPSInsights cannot know exactly whether the data are handled fully by the other components. To overcome these, there should be a mechanism to maintain a consistent view of what has been processed successful by the system. Therefore, we have to introduce a transaction guarantee to GPSInsights to ensures that either all output data from Spark Streaming are logged to MongoDB or the arriving data are reprocessed.

To achieve this guarantee, first, we implemented a new Spark Streaming's Receiver by using the Spark Streaming's "receiverStream" API and the Kafka's Simple Consumer API. Instead of handling only the latest data, the new Spark Streaming Receiver can specify the start position of the offset for each partition at the beginning of every batch interval. It can also get the extra information of each record including its offset, id of the partition which it belongs to. Second, we created a MongoDb database with three different collections, namely "Transactions", "Records", and "OffsetRanges". The "Transaction" consists of documents having three fields: id, timestamp and status. The status field can accept two value: "BEGIN" means the beginning of a transaction and "FINISH" expresses the end of the transaction. "Records" contains documents which are the information of the analytic results from Spark Streaming and an id of the transaction. "OffsetRanges" includes documents which hold the information of an offset range of the records packaged into the current batch, and the id of the transaction.

The detail implementation is described as follows. Before sending to Spark Streaming's Receiver, each record in Kafka will be attached with its offset and partition's id which it belongs to. Using Accumulator API [20], we can find the offset range of each Kafka's partition in the current batch. When finishing handling this batch and before logging the result data to MongoDb, we create a new document with "Begin" status in "Transactions" collection and get its id. We then create a new document in "OffsetRanges" collection with the offset range and the id. Next, we send the result data to "Records" collection, attaching the id. Finally, after the last record is written successful in MongoDb, we change the status field of the transaction to "Finish", and the current batch is handled successful. During running, if the system fails and then recovers, it will query MongoDb for the last document in "Transactions" collection. If the value of the status field is "Finish", it means the process of handling the last batch was succeeded. By contrast, we will use the transaction id to get the relevant document in "OffsetRanges" collection, and use the first offset in each ranges (the number of range is equal to the number of partition of the Kafka' topic that we are consuming) to recomputed the data.

4. SCALABLE MAP MATCHING

In this section, we demonstrate how GPSInsights handles the map matching job in a scalable way. Map matching job aims at associating location data to the road network on a digital map. This is the first and the required step for many location data mining algorithms.

4.1 General strategy

GPSInsights uses Open Street Map (OSM) engine [14]. The Open Street Map's raw data consist of a mass of tag almost covering the whole world. Every node tag has some tags inside to determine its attributes (e.g. type, way name, coordinate, ..). Way tag contains one or more node tags that used to define the shape of it. It is defined that link is a section of road between intersections. In most digital maps, a real road (or way) is digitized and is described with a set of many straight lines. Vertices are points which separate these straight lines, and each straight line is a segment. For example, the road AD in Fig. 5 is formed by 3 intersections (black points), 2 vertices (white points), 2 links (AD, DE) and 3 segments(AB, BC, CD).

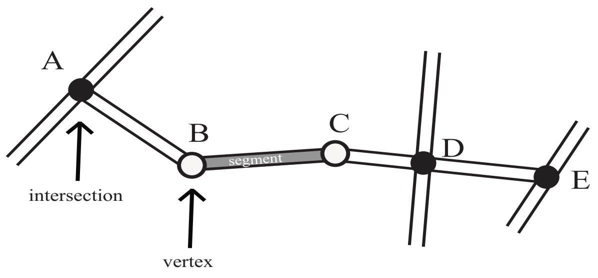

Figure 5: Road network composition

As location data captured by GPS device expose to distance errors (its standard deviation is around 3 meters at best), map matching is a difficult task. GPSInsights solves this task by enriching the OSM data with a lot of intermediate nodes (beside the nodes declared in OSM) to every road. Thus, each particular location point will be assigned to the road that the closest point belong to. To find closest point, we use KD-tree [15].

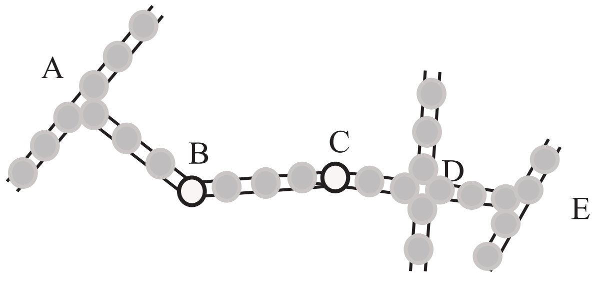

Figure 6: Adding intermediate nodes into road segments

Technically, adding intermediate nodes (illustrated in Fig. 6) is done as following. Let's suppose we add a point B into a segment AC in order that the distance between this point and A is d_{AB} ($d_{AB} < d_{AC}$). The coordinate of the point B is determined as following formula:

$$latitude_B = latitude_A + (latitude_C - latitude_A) * \frac{d_{AB}}{d_{AC}}$$

$$longitude_B = longitude_A + (longitude_C - longitude_A) * \frac{d_{AB}}{d_{AC}}$$

where d_{AC} is the metric distance between the point A and the point C, calculated on *Haversine* formula [6].

While adding new points, we also remove all intersections and finally build a KD-Tree based on those nodes. Note that a node in the Kd-Tree consists of a coordinate and information of a link which associates it. Therefore, by taking a GPS point into the Kd-Tree and using the vertex-based map matching, we can determine nearest nodes as well as a link it is matched to.

4.2 Detail algorithm

After getting input data from the distributed publish-subscribe messaging service (Apache Kafka), the kd-tree is built and then be broadcasted to all the computing nodes (servers) of the streaming processing engine (Storm or Apache spark). By doing this, in the case that our system uses Spark Streaming, we can keep a read-only variable cached on each machine instead of shipping a copy of it with every tasks, this will help improving the system's performance.

As we have a parallelized collection formed, our algorithm will be divided into two phase: a data mapping phase and a data collecting phase.

4.2.1 Data Mapping Phase

This phase is responsible for matching coordinate of every transportation record q into a concrete segment of the road. The algorithm considers the following attributes of q: the latitude, the longitude and the speed. These records will be evaluated by kd-tree cached on each machine to find out which road the record q belongs to. Because of the fact that coordination GPS sent from satellites will have some deviations in comparison with the real coordinate, we have to choose a threshold distance in order to determine whether q belongs to the road or not. If the distance is smaller than the threshold distance, it will be much easier for us to conclude that q belongs to the road segment and vice versa. As a result, our algorithm can ignore q that its distance within the road segment is too far from. Therefore, we can enhance the accuracy of our algorithm to some extent.

4.2.2 Data Collecting Phase

In data mapping phase, the transportation data on each time step are transferred to the distributed stream processing engine will be matched into road segments in-parallel. Our algorithm then collects the output form all the computing nodes to achieve the final statistic results. In this phase, all the transportation data belong to the same road ID is aggregated. Hence, we can obtain <key, value> pairs with the key being the road segment's ID, the value that contains the number of vehicles moving in that road segment and the sum of their speeds v_s. Then, we calculate the average speed of every links (a section of a road between two intersections, as we described in section 4.1) using the following "space mean speed" formula which we will give an detailed explanation in the *appendix* section.

$$v_s = \frac{N}{\sum_{i=1}^{N} \frac{1}{v_i}}$$

where v_i is the spot speed of i^{th} vehicle, and n is the number of vehicles.

5. EXPERIMENTS

Our data include about 12,565,521 GPS records collected by vehicles equipped with the GPS receiver from 22/03/2014 to 22/04/2014 in Ho Chi Minh city. Every record consists of the speed, the GPS coordinate and the state of the vehicle. The following table 5 shows the format of the data.

time_stamp	car_id	lon	lat	speed

GPSInsights system is set up on a cluster of HPCC super computer, consisting of 4 nodes, one master node and three slave nodes. Each cluster node is equipped with a 8-cores Intel Xeon 2.6GHz CPU, 32GB memory. We benchmarked GPSInsights configured with Spark Streaming against GPSInsights configured with Apache Storm and compared with the common system using Geomesa for map matching (based on the ability of querying K-nearest neighbor search of this database). For all experiments, the results are reported by averaging three runs.

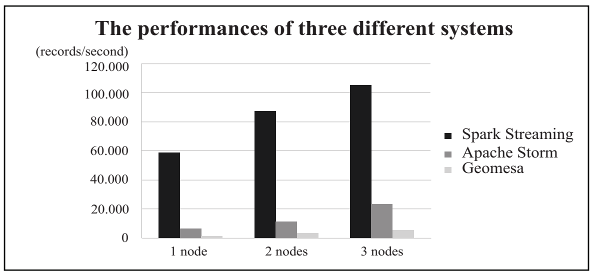

Figure 7: Measuring processing performance in term of completion time

Figure 7 presents the performance of GPSInsights using Spark Streaming, one using Apache Storm and the Geomesa system for handling the transport records with the number of slave nodes from 1 to 3. In this experiment, the processing time of the system is measured from the time the streaming processing engine (Spark Streaming or Apache Storm)/Geomesa receives the data from Kafka, until the analytic data are sent to the storage components successful. It is clear that the number of slave nodes was increased, the execution time of the system reduced steadily. However, with about 52.000 handled records/second (1 node), 86.000 (2 nodes) and more 105.000 (3 nodes), our system using Spark Streaming saw the highest performance, being around 9 times faster than using Apache Storm, while the least one is the Geomesa system, just dealing with around 5000 records per second with 3 slave nodes. The Spark Streaming system achieving the best performance is mainly due to taking the advantage of the in-memory batch handling strategy, we can execute the nearest neighbor search directly in memory by using the map-matching algorithm shown in Section 4.

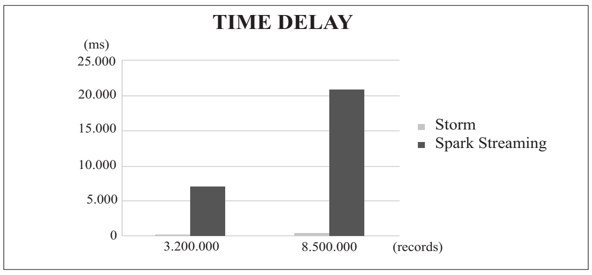

Figure 8: Measuring data processing latency

Figure 8 shows the processing latency of GPSInsighs configured with two different frameworks (Spark streaming and Storm). Although the one using Spark Streaming is so much faster than using Storm, GPSInsights using Storm showed lower latency. Because Storm processes incoming events one at a time, so that its processing latency could be sub-second. Otherwise, after collecting the input data into a batch, Spark Streaming then divide it up into partitions and sent them to slave nodes. Thus, the processing latency of Spark Streaming depends on the batch interval (if the batch interval is 5 seconds, it means that Spark has to wait 5 seconds before starting processing the first batch) and the number of records packed in the batch (the time for sending partitions to slave nodes is directly proportional to the quantity of records).

6. RELATED WORK

PostGIS [16] is a spatial database extender for the PostgreeSQL relational database, adding support for geographic objects allowing location queries. It is undeniable that PostGIS completely meets most of the spatial-temporal query and there are a large number of software products that use PostGIS [22]. However, the spatio-temporal data sets have seen a rapid expansion in volume and velocity due to the rapid increase of means of transport and GPS device, which is about hundred millions and more records per day. Whereas PostGIS scalability is limited to small size clusters, GPSInsighs leverage Geomesa as temporal-spatial storage is efficient for tackling big transport data set.

There are a number of studies on matching GPS observations on a digital map. Several map matching mechanisms consider only the relationships between GPS data and the digital map [13]. Recent works [7] leverage the topology of the road network and the GPS data trajectory to improve accuracy. However, non of those works designed for large-scale datasets.

7. CONCLUSION AND FUTURE WORK

This paper presents GPSInsights, a scalable, extensible and reliable system for continuously storing and processing massive amounts of vehicle location data. We described the main components of the system, explained our design principles, and demonstrated the scalability of the system with the map-matching challenge. We also described our solution to insure that the stream of vehicle data is handled atomically and reliably, by rewriting the Spark Streaming's receiver. In addition, through conducting experiments with real vehicle datasets, we proved that GPSInsights is able to address massive GPS vehicle data in a scalable fashion.

In the future work, we intent to pursue the system in the following directions. First, we aim at introducing into GPSInsights an advance map-matching algorithm with higher accurate on low-sampling-rate vehicle data. Second, we research real-time traffic jam prediction through mining stream datasets.This opens room for smart applications such as: to recommend mobile users the fastest route path that is calculated in real-time.

8. REFERENCES

[1] Apache_Accumulo. https://accumulo.apache.org/.
[2] Apache_Kafka. http://kafka.apache.org/.
[3] Apache_Storm. https://storm.apache.org/, 2015.
[4] U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. *A Case for Time-Dependent Shortest Path Computation in Spatial Networks.* Advances in Spatial and Temporal Databases Lecture Notes in Computer Science, 2010.
[5] Fox, Anthony, Eichelberger, Chris, Hughes, James, Lyon, and Skylar. Spatio-temporal indexing in non-relational distributed databases. In *2013 IEEE International Conference on Big Data*, pages 291–299. IEEE, 2013.
[6] Haversine_Formula. https://en.wikipedia.org/wiki/haversine_formula.
[7] Y. Jae-seok, K. Seung-pil, and C. Kyung-soo. The map matching algorithm of gps data with relatively long polling time intervals. *Journal of the Eastern Asia Society for Transportation Studies*, 6:pp. 2561–2573, 2005.
[8] JavaSpringMVC. http://projects.spring.io/spring-framework/.
[9] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning Spark.* O'Reilly Media, 2015.
[10] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. *NetDB*, 2011.
[11] W. H. Lee, S. S. Tseng, J. L. Shieh, and H. H. Chen. Discovering traffic bottlenecks in an urban network by spatiotemporal data mining on location-based services. *IEEE Transactions on Intelligent Transportation Systems*, 12(4):1047–1056, 2011.
[12] MongoDB. https://www.mongodb.com.
[13] Noh and Kim. A comprehensive analysis of map matching algorithms for its. *Hongik Journal of Science and Technology*, 9:pp. 303–313, 1998.
[14] Open_Street_Map. www.openstreetmap.org.
[15] M. Otair. Approximate k-nearest neighbour based spatial clustering using k-d tree. *International Journal of Database Management Systems*, 5(1), 2013.
[16] PostGis. http://postgis.net/.
[17] Seen T. Allen, Matthew Jankowski, and Peter Pathirana. *Storm Applied: Strategies for real-time event processing.* 1st edition, 2015.

[18] Shvachko, Konstantin, Kuang, Hairong, Radia, Sanjay, Chansler, and Robert. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[19] Spark_Streaming. http://spark.apache.org/streaming/, 2015.

[20] Spark_Streaming-Accumulator. https://spark.apache.org/docs/latest/programming-guide.html#accumulators-a-nameaccumlinka.

[21] WebSocket. https://www.websocket.org/.

[22] Wikipedia. User section: http://en.wikipedia.org/wiki/postgis.

[23] C. M. Wu, Y. F. Huang, and J. Lee. Comparisons between mongodb and ms-sql databases on the twc website. *American Journal of Software Engineering and Applications.*, 4(3):35–41, 2015.

[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[25] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438. ACM, 2013.

[26] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

APPENDIX

A. SPACE MEAN SPEED

The formula in *DataCollectingPhase* (4.2.2) section could be determined as follow:

Let t_i is the time the vehicle having a speed v_i take to complete a link having a length D. So

$$t_i = \frac{1}{v_i}$$

And the average speed v_s of all vehicles traveling in the link is their total distance divided by their total time.

$$v_s = \frac{\sum_{i=1}^{N} D}{\sum_{i=1}^{N} t_i}$$

It's equal to:

$$v_s = \frac{N * D}{\sum_{i=1}^{N} \frac{D}{v_i}} = \frac{N}{\sum_{i=1}^{N} \frac{1}{v_i}}$$

B. THE VISUAL DISPLAY SYSTEM

The visual display system is an application of GPSInsights that shows real-time the result of road average speed computation for users. This application is built based on Java Spring MVC framework [8] and the Open Street Map data [14]. It pulls the data from the distributed result queue through the connector module illustrated in the Figure 9.

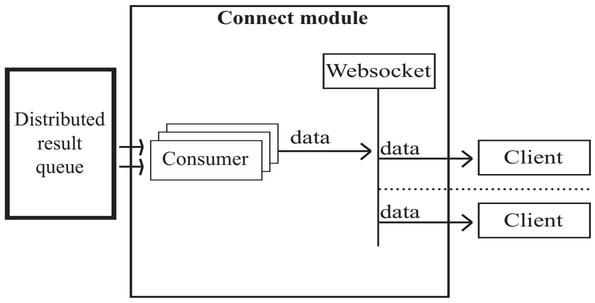

Figure 9: GPSInsights connector module

There are two main components in this module: Kafka's consumers and WebSocket [21]. WebSocket is responsible to deliver a two-way communication channel. The application map is depicted in Figure 10. Green lines express the road having the average speed 30 km/h and over, yellow lines show the average speed from 15 km/h to 30 km/h, orange lines are for 5-15 km/h and the red lines mean having the traffic jam in roads.

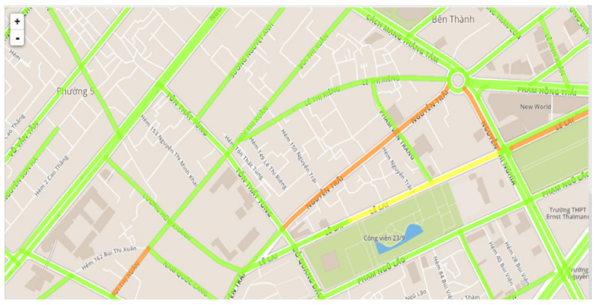

Figure 10: Map with road average speed updated in real-time

C. MAP MATCHING PSEUDOCODE

The below pseudocode illustrates the map matching algorithm installed in our system with Spark Streaming.

```
// kdTree: a KD-Tree build up from nodes of OSM data
// input: a list containing data queried
// output: a list of tuple2<key, value> with key being a roa
// and value being a number of vehicle and average speed
// scc: a Java Spark Context variable
// lines: distributed datasets
// broadcastVal: keep a read-only variable of the kdTree
// currentPoint: GPS vehicle
// nearestNode: nearest node of current point
// distance: distance between currentPoint and nearestNode
// listRoad: a <key,value> generated after mapping lines wit
// being a roadID, value being a iterable of vehicleID and s
// result: a <key, value> generated after mapping listRoad with key
// being a roadID, value being vehicle volume and average speed
Build up the kdTree
Broadcast<KDTree<GeoName>> broadcastVal = scc.broadcast(kdTree);
JavaRDD<String> lines = scc.parallelize(input).cache();
JavaPairRDD<String, Iterable<String>> listRoad = lines
    .mapToPair(New PairFunction<String, String, String>() {
        Public Tuple2<String, String> call(String p1)
            Throws Exception {
            nearestNode = broadcastVal.value().
                              findNearest(currentPoint);
            If distance > threshold then
                roadID = Null;
            Else
                roadID = nearestNode.toString();
                vehicleID = currentPoint.ID;
                speed = currentPoint.speed;
                attributes = vehicleID + speed;
            end If;
        Return New Tuple2<String, String>(roadID, attributes);
    }).groupByKey().cache();
JavaPairRDD<String, String> result = listRoad
    .mapToPair(New PairFunction<Tuple2<String, Iterable<String>>,
               String, String>() {
        Public Tuple2<String, String> call(Tuple2<String,
               Iterable<String>> tuple1) Throws Exception {
            numberOfVehicle = Count the number of element
                              in iterable value of listRoad;
            averageSpeed = Calculate average speed from
                              iterable value of listRoad;
            statistic = numberOfVehicle + averageSpeed;
        Return New Tuple2<String, String>(roadID, statistic);
    });
Collect result
    output = result.collect();
end Collect;
```