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

[Extended Abstract]

Tobin Institute for Clarity in Documentation
1932 Wallamaloo Lane
Wallamaloo, New Zealand
trovato@corporation.com

G.K.M. Tobin Institute for Clarity in Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
webmaster@marysville-ohio.com

Lars Thørväld The Thørväld Group
1 Thørväld Circle
Hekla, Iceland
larst@affiliation.org

## 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. [10] presented a data mining algorithm to discover traffic bottlenecks. Demiryurek et al. [3] 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 in must be equipped with a standardised global positioning system (GPS) (black-box) which reports geo-location, speed and direction every 30 seconds to a centralised data center. 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 (PB)) 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 [ref]) 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 ( PB) while supporting real-time analytics. We demonstrates 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 components of GPSInsights. In Section 4, we establish a simple demonstration map matching algorithm. Section 5 presents experimental results for the algorithm presented in Section 4. 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 [ref]:

- 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 information and timestamp in which the data is 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.

# 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.

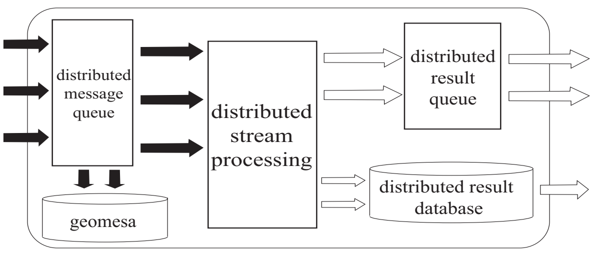


**Figure 1: The architecture of the system**

### *3.0.1 Apache Kafka.*

Apache Kafka [8, 9] 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 [ref]. 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, the producer that publish messages can distribute messages over multiple partitions of a topic in a round-roubin fashion, or it can distribute according to some semantic partition functions.

Kafka achieve 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 follower will be automatically promoted to be the new leader (using the well-known distributed leader election algorithm). In production, every Kafka node acts as leader for some partitions and 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 agregating in-coming location data and route analytics results to the appropriate layers (E.g. to the storage and to the end-user applications).

### *3.0.2 Spark Streaming and Storm.*

Spark Streaming [5, 19] and Apache Storm [16, 18] are the most popular open-source frameworks for distributed stream processing. In 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 [12] for acting on data stream. At the core of Spark Streaming is the concept of discretized abstraction (D-streams) [14, 13], that considers in-coming records as a series of deterministic batchs on small time intervals. Each batch is treated as a reslient 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 computations within RDD are recorded in logs (called lineage). Third, temporary data is 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 the input data which is usually at immutable Hadoop file system HDFS [17], 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 (so storm's processing latency could be sub-second, while Spark Streaming reaches a latency 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 are received by a specialized component called a "spout". Then the spout immediately passes the data to another component called a "bolt". In a bolt, the data will be transformed in some way, and the bolt either sends it to some sort of storage or passes it to some other bolt. In general, a 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 vertial: more data means bigger server. One way to scale relational databases across multiple server 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 [15] is a document store with the posibility 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 colume 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 [2], 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 [4] 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 a cluster to leverage parallelism, thus enables efficient storing, querying, and transforming large spatio-temporal data. Geomesa is like PostGIS [ref] 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 necessery. 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 realibility

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 a 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 continue handling.

However, GPSInsights have to deal with unrealiability 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.

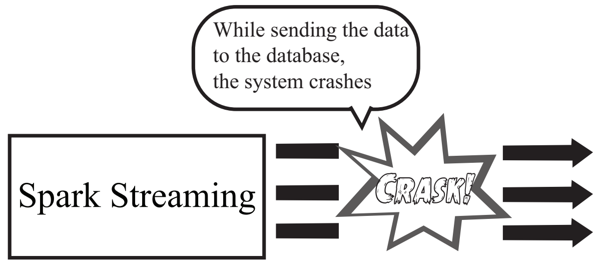


**Figure 2: The first problem**

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. Therefore, Spark Streaming supposed it received the data, but Kafka supposed that the messages was not sent successfully and would resend repetitively.

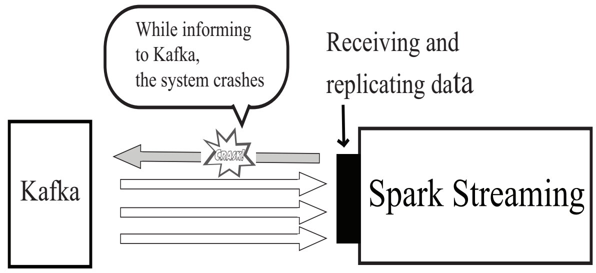


**Figure 3: The second problem**

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 /citekafkasimpleapi. 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 IMPLEMENTED INSIDE GPSINSIGHTS

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 firstly required step for many location data mining algorithms.

| time_stamp | car_id | lon | lat | speed |
|---|---|---|---|---|

## 4.1 Algorithm: Road Reverse Geocode Algorithm Using K-D Tree.

Before going to the details, vertices, links and segments should be defined clearly. Link is a section of road between intersections illustrated by black points in Fig. 4. In most digital maps, a real road is digitized and is described with a set of many straight lines. Vertices are white points in Fig. 4 which separate these straight lines, and each straight line is a segment. For example, the road AD in Fig. 4 is formed by 3 intersections, 2 vertices, 2 links (AD, DE) and 3 segments(AB, BC, CD).

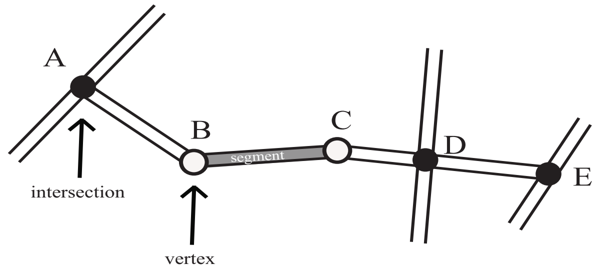


**Figure 4: The composition of the road network**

The clear definition of the composition of the road network helps us to understand map matching algorithms which in our problem are supposed to assign every GPS point to the relevant links of a digital map.

There are many map matching algorithms which have been developed to tackle this, including vertex-based or segment-based map matching, map matching using the geometric relationship or the network topology the data history and so forth. But the most of them are studied for the GPS data with short and relatively long polling time intervals (less than 5 minutes), so not appropriate for our data (about 15 minutes). Also, this paper does not focus on the map-matching problems, we only use the vertex-based map matching for simplicity and high-performance to illustrate the scalable map-matching of our system.

This method could be simply described as follows. With a GPS point, we will find a corresponding vertex in the digital map such that the distance between them is shortest, the point then is matched to the link containing this vertex. However, looking for one most relevant point amongst a variety of point take many time in the normal way, this will affects our system performance. So we use kd-tree as the tool to tackle this problem [11]. It is an efficient searching method to quickly find nearest points and this algorithm only takes O(log n) average time per search in a reasonable model.

The vertex, intersection and link data using to build kd-tree are exported from the Open Street Map data, an open-source digital map is contributed by many users across the world.

But the map matching using the distance of point-to-point has problem of abnormal result due to several reasons.

**The composition of the digital map.** As shown in Figure 5, the GPS point belongs to Link 1 was matched to Link 2, since the distance between the GPS point with P4 is small than the points of Link 1. To overcome this problem and to increase the accuracy of the map matching, we add some equidistant vertices in segments illustrated in Figure 6, this

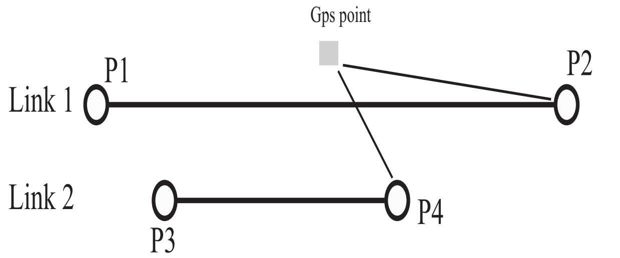


**Figure 5: Matching the GPS point to the wrong link**

will be dealt as follow:

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].

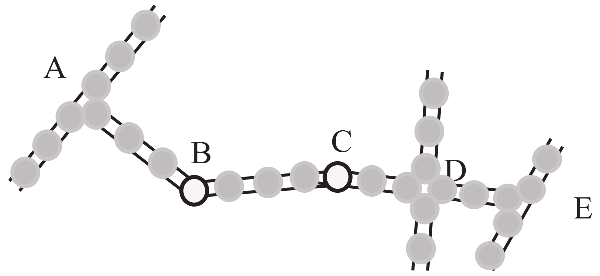


**Figure 6: Add some equidistant vertices in segments**

**The problem in the vicinity of an intersection.** This problem can be resolved by adding some points with particular radius around the intersection and not using the intersection data to build kd-tree. This is shown in the figure 6), you can see the A, D, E intersections being replaced by some adding vertices.

Apart from, to enhance the accurate of our map matching algorithm , we also choose the particular threshold distance in order to determine whether an GPS point really belong to the road link or not. If the distance from the GPS point to the nearest vertex is smaller than the threshold, it will be much easier for us to conclude that the GPS point really belongs to the road links, on the contrary, it will be removed.

#### 4.1.1 *The scalability.*

In this section, we focus on explaining how our proposed algorithm is used in our system and why having the scalable ability. After getting input data from the distributed publish-subscribe messaging service (Apache Kafka), now, we review how the distributed stream processing (Spark Streaming and Apache Storm) is used in our algorithm. Firstly, we build up a kd-tree with the preprocessed OSM data and then transfer its copies to the nodes in our system. In the case that our system uses Spark Streaming, with calling Spark's broadcast API, we can keep a read-only variable cached on each machine instead of shipping a copy of it with tasks, this will help improving the system's performance. However, providing every node a copy of the kd-tree will increase a memory and construction time.

With the kd-tree copies on every nodes in our system, matching the input GPS data to the links will be executed in parallel. In other words, a master node of our system will divide the input GPS data on each time step into small parts, then sending them to the nodes holding the kd-tree copy for map-matching. After map-matching phase, the results will be grouped the matched pairs by road segment's ID and returned to the master node. Obviously, when the system is extended by adding some nodes, the map matching algorithm automatically is packed and sent to the new nodes, so it is very suit for the scalable system like our system.

## 5. EXPERIMENTS

### 5.1 Dataset

Our data include about 12,565,521 GPS records collected by vehicles equipped with a GPS receiver from 22/03/2014 to 22/04/2014 in Ho Chi Minh city. Every record consists of speed, GPS coordinate and state of the vehicle and the period of time between two records is 15 minutes. The following table shows the format of the data.

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 With this configuration, we have evaluated the performance of GPSInsights, using the dataset shown in section 4.1 to simulate the stream data. Firstly, We compared the running time of the system on the different number of records with various numbers of slave nodes. Secondly, we benchmark the system using Spark Streaming against the system using Apache Storm. For all experiments, the results are reported by averaging three runs.

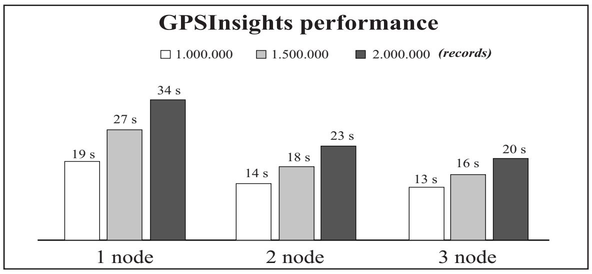


**Figure 7: GPSInsights performance**

Figure 7 presents the performance of employing GPSInsights for handling three different quantities of input 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 Spark Streaming pull the data from Kafka into a batch, until the analytic data of the batch are sent successful to the storage components. It is clear that the number of slave nodes was increased, the execution time of the system reduced steadily, this is thanks to the parallel procedure of Spark. Besides, we also installed a common system with Geomesa for map-matching, based on the ability of querying K-nearest neighbor search of this database, in order to have a comparison with our system. With 1.200.000 records, after 10.000 seconds running, there was no signs of stopping of the system. The performance gap between two system is mainly due to taking the advantage of Spark Streaming in our system, we can execute the nearest neighbor search directly in memory by using the map-matching algorithm shown in section 4.2.

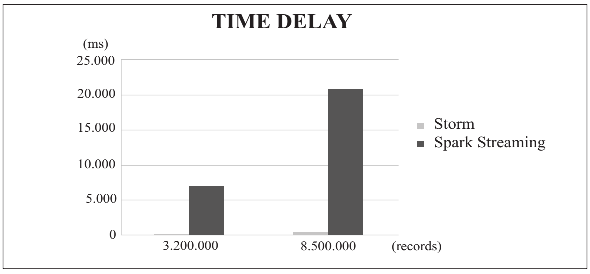


**Figure 8: Time delay**

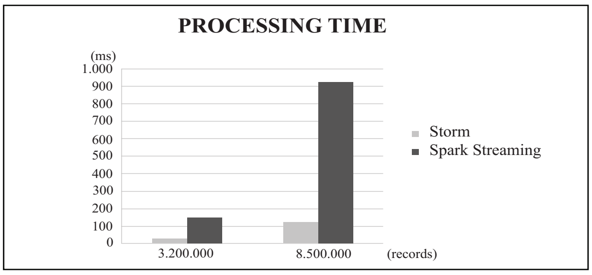


**Figure 9: Processing time**

Figure 8 and Figure 9 show the time delay and the processing time of two systems using two different frameworks as the streaming data processing engine for performing the map-matching algorithm in 3.200.000 and 8.500.000 records of the input data. The system using Storm beats one using Spark Streaming in term of the lag. 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. So 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). However, with the in-memory batch handling strategy, the time for processing the data of Spark Streaming is so much faster.

# 6. CONCLUSION AND FUTURE WORK

This paper presents GPSInsights, a scalable, extensible and reliable system for continuously processing the massive amounts of vehicle data. We described the main components of the system, explain why to choose them, and the map-matching algorithm being inside of. We also described the method to improve the system, insuring the vehicle data which are handled exactly one, by rewriting the Spark Streaming's receiver and building the simple NoSQL transaction for MongoDb. In addition, through conducting the vehicle data statistic, we proved the GPSInsights' potential to address an increasing number of the problem classes relating massive GPS vehicle data.

In the future work, we intent to pursue the system in three directions. Firstly, We will use the advance map-matching algorithm with higher accurate on low-sampling-rate vehicle data, about 15 seconds or less. Secondly, GPSInsights will be installed new algorithm for predicting future traffic conditions based on real-time data.Finally, the system will be improved to find the fastest path depending on the travel times of each road segment.

# 7. REFERENCES

[1] Accumulo. https://accumulo.apache.org/.

[2] Chieh M. Wu, Yin F. Huang, and John Lee. Comparisons Between MongoDB and MS-SQL Databases on the TWC Website. *American Journal of Software Engineering and Applications.*, 4(3):35–41, 2015.

[3] 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.

[4] 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.

[5] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning Spark.* O'Reilly Media, 2015.

[6] Haversine_Formula. https://en.wikipedia.org/wiki/haversine_formula.

[7] JavaSpringMVC. http://projects.spring.io/spring-framework/.

[8] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. *NetDB*, 2011.

[9] A. Kafka. http://kafka.apache.org/.

[10] 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.

[11] M. Otair. Approximate k-nearest neighbour based spatial clustering using k-d tree. *International Journal of Database Management Systems*, 5(1), 2013.

[12] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion 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.

[13] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion 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.

[14] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion 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.

[15] MongoDB. https://www.mongodb.com.

[16] Seen T. Allen, Matthew Jankowski, and Peter Pathirana. *Storm Applied: Strategies for real-time event processing*. 1st edition, 2015.

[17] 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.

[18] A. Storm. https://storm.apache.org/, 2015.

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

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

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

# APPENDIX

## A. SPACE MEAN SPEED

The formula in *DataCollectingPhase* (**??**) 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

We also made the system to show periodically the statistic result for users. This display system is built based on Java Spring MVC framework [7] and the Open Street Map data, getting the data from the distributed result queue through the connect module illustrated as the following figure:

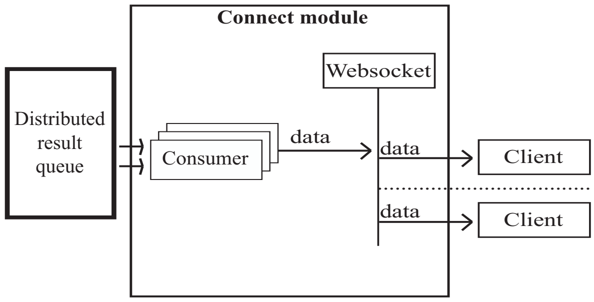


**Figure 10: The connect module**

There are two main components in this module: Kafka's consumers and WebSocket [21]. Using WebSocket is in order to send the data to many clients in a short period of time. While the purpose of choosing the consumer is to minimize the waiting time of the data processing engine when pushing the final statistic results to the connect module, by taking advantage of read/write in parallel of Spark Streaming/Apache Storm and especially Apache Kafka. After receiving the data from the connect module, the display system will draw them as follow:

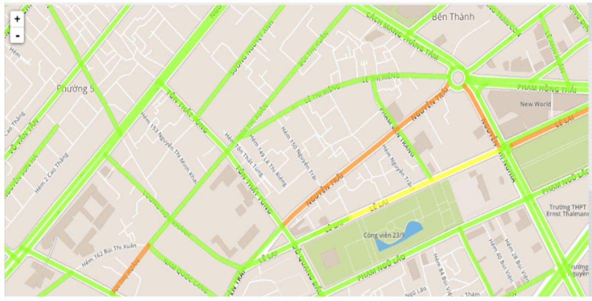

**Figure 11: Showing the statistic results for users**

Where the green line expresses the road having the average speed 30 km/h and over, the yellow one shows the average speed from 15 km/h to 30 km/h, the orange one for 5-15 km/h and the red line means having the traffic jam in the road. Also, the system will update automatically as soon as receiving the new results.