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

[Extended Abstract]

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

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

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

ABSTRACT

Intelligent Transport System (ITS) has seen growing interest in collecting location-based data of transport vehicles in order to build up a real-time traffic management system. Managing those data faces big data challenges. It has big volume and big velocity characteristics as the data is collected continuously every short period of time from hundred thousand of vehicles. Current ITSs rely on relational database and sequential processing engine cannot scale. In this paper we propose GPSInsights, a framework that can manage and process massive GPS vehicle data effectively. GPSInsights is built on scalable compopents such as large-scale storage and distributed processing engine. We demonstrate our framework with a scalable map matching implementation and perform several experiments with real 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-based 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. [7] a data mining algorithm to discover traffic bottlenecks. Demiryurek et al. [2] proposed an online computation of optimal traffic route based on traffic data. 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 message queue: As the location data come continuously from a huge number of source vehicles, this component is designed to combine the multi-source data, and put them in their chronological order. It has to store and replicate the large input data dispersedly on a large cluster for high-throughput, and 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 on open-source components with custom plugins to satisfy our design goals . By leveraging existing components, we can develop GPSInsights quickly and focus on the scalable aspect of the entire system. A part of our contribution is then about designing the system overall, carefully choosing the right components, and running the experiments with real datasets.

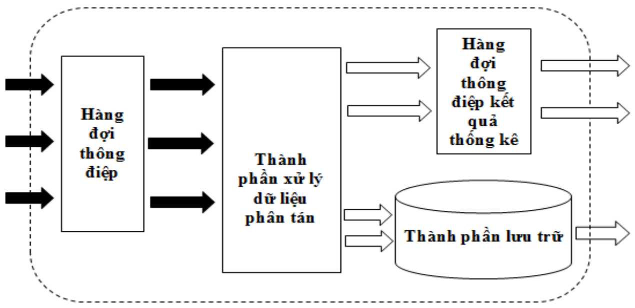
Hàng đợi thông điệp
Thành phần xử lý dữ liệu phân tán
Hàng đợi thông điệp kết quả thống kê
Thành phần lưu trữ

Figure 1: The architecture of the system

3.0.1 Distributed message queue: Apache Kafka [ref].

Apache Kafka is the distributed publish-subscribe messaging. Kafka stores the data in categories called topic. A topic will be divided into one or more partitions which be distributed over nodes of a cluster (each partition is an ordered, immutable sequence of messages). It allows store the amount of the data larger than the capability of any single machine. Internally, each record of the data called a "message", uniquely specified by an id number called "offset" which is its ordinal number in a partition. Reading to and writing data in topic are very fast, thanks to paralleling operations and several useful techniques installed to maximize the performance. According to Kafka's official website, a single Kafka broker can handle hundreds of megabytes of reads and writes per second from thousands of clients. Kafka is designed to allow transparently and easily expanded without downtime. Kafka also can replay any message or set of messages given the necessary selection criteria. In addition, partitions are persisted on disk and replicated within the cluster to prevent data loss. So Kafka offers strong scalable, durability and fault-tolerance.

3.0.2 Streaming data processing engine: Spark Streaming [ref] and Storm [ref].

Spark Streaming and Apache Storm now are the most popular open-source frameworks for distributed stream processing. However, there are important differences in their architectures as following.

Spark Streaming is a module of Apache Spark for acting on data stream as soon as it arrives. It provides an abstraction called Dstreams (discretized streams). A Dstream is represented as a continuous sequence of data arriving over time. More detail, a set of data arriving at each time step, received by a component called "receiver", will be packed in a batch called an RDD, so each Dstream is a continuous sequence of RDDs. An RDD is simply an immutable distributed collection of objects. We cannot change any property of an object in RDD. Using Map-Reduce model, Each RDD is split into multiple partitions, which may be processed by "executers" on different nodes of a Spark Streaming cluster. Each operation on partition of RDD is also executed in memory, so running programs on Spark Streaming is very fast. According to Spark's official website, by comparison with the time of executing a logistic regression in Hadoop, running time in Spark is 100 times faster. Spark's RDDs also offer a strong fault-tolerance , because the input data are replicated to other nodes of the cluster (In Spark 1.1 and earlier, received data was replicated in-memory only, so it could also be lost if Spark Streaming crashed. Since Spark 1,2, received data are also persisted to a reliable filesystem like HDFS so that they are not lost anymore), and an RDD remembers the lineage of deterministic operations that were used on that data to create it. If any partition of an RDD is lost due to a node failure, as long as a copy of the input data is still available, then that partition can be re-computed from it using the lineage of operations. So that Spark Streaming provides better support for processing each batch exactly once.

Instead of batching up events that arrive within a short time and then processing them like 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 an other 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 short, 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.

In GPSInsights, we mainly focus on Spark Streaming. The purpose of installing Storm as the streaming data processing engine is to compare the performance of the system when using the two frameworks to solve the same problem. The results of the comparison are expressed in the Experimental Evaluation section.

3.0.3 Distributed result queue: Apache Kafka.

Providing a very high-throughput, Apache Kafka is relatively suitable for this position. The data processing engine will use Kafka's producer to push the statistic result data to a Kafka's topic in parallel. According to the benchmark experiments of the Kafka's developing team, Kafka managed to write more 2 million records (approximately 193.0 MB) per second on three cheap machines (https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines). Hence, with a suitable hardware configuration, the data processing engine can just spend a negligible time for writing the result data to the Kafka's topic. The data processing engine then continues the next input data, while the connect module uses Kafka's consumer to pull that data and send to every visual display systems of clients. This helps the data processing engine minimize the waiting time of sending the result data to external component.

3.0.4 Distributed spatio-temporal database: Geomesa [ref].

GeoMesa [6] is an open-source, distributed, spatio-temporal database built on top of a column-family oriented distributed database called Accumulo [1]. GeoMesa uses a very flexible strategy to linearize geo-time data and distribute the data across the nodes in a cluster to leverage parallelism in computationally intensive queries. It enables efficient storing, querying, and transforming capabilities for large spatio-temporal data. Furthermore, GeoMesa supports open OGC APIs, which facilitates transition from other databases to this database.

Accumulo is a NoSQL database that develops distributed and sorted key/value pairs store based on the Google's Big Table design [4]. However, instead of storing data in a distributed file system called GFS as Goolge's BigTable, Accumulo use HDFS as its data storage. It is designed to include an active Master and many Tablet Servers. A TabletServer manages some tablets which are located on a datanode of Hadoop [11, 3].

Configuration and status information of Accumulo is stored in Zookeeper [15, 5] to monitor changes in the cluster. With ZooKeeper, Accumulo can configure multiple Master processes. Whichever process gains a Master Zookeeper lock first will be the active Master. The inactive Masters will constantly track the active Master's Zookeeper lock, waiting to obtain the lock in the case the active Master fails. Besides, whenever clients wish to locate the data they seek, Zookeeper will navigate them to the right servers.

Among a number of other column family oriented databased which have been developed and released, GeoMesa uses Accumulo because of an ability to add server-side programming via a feature called iterators [12]. The Accumulo iterators provide a sorted view of the key-value pairs in the table and allow to search across the row or column family dimensions. Through filtering and transforming data function are provided by user-defined iterator, we could require the database to return the desired data or add a combiner to produce statistics on the filtered rows.

Geo-Time Data indexing strategy. In a NoSQL database such as Accumulo, it has only a single index that is built atop a constraint that all records are ordered lexicographically. It is a problem when we want to map from three dimensions (lat, lon, time) to one (lexicographical ordering of keys in a table). So encoding geo-time information in keys is an essential purpose of a geospatial indexing scheme. The location of the point is represented as a 7-character string using Niemeyer's encoding from a 35-bit geohash and the datetime information is encoded as a string of the format 'yyyyMMdd'. These two string , one for location and one for datetime, are used to build an index entry by the indexing strategy of GeoMesa.

Besides, GeoMesa also uses Space-filling to execute queries efficiently because we only perform a minimum number of steps to seek relevant entries instead of traversing all ones. A common example is Geohash, which uses interval halving on latitude and longitude to build up a bit-string of alternating dimensions: the first bit is 0 or 1 showing whether the point interest is less than 0.0 degrees longitude or not; the second bit is for latitude; the third bit is for the second sub-division of longitude and so on. A query plan now is simply a matter of finding which subsets of the space-filling curve fall within the polygon of interest.

Geomesa data layout. The space-filling curve has a drawback when it used to index geo-time data: a hotspot will appear if all of the data satisfying a given query are contained in a single tablet server. To address this problem, GeoMesa index schema-formats default to add a random bin number (between 00 and 99) at the beginning of an index entry to help uniformly distributing data across tablet servers. This prevents any tablet server from being over-tasked and help to balance the work-load across a cluster in dealing with a larger number of queries.

Query Planning of Geomesa is described as follow. First, the query polygon is decomposed into GeoHash ranges to scan. Second, the temporal bounds are used to refine the row range. Finally, we could exactly determine every rows that must be inspected by using the bin number.

3.0.5 Distributed analytic result database: MongoDb [ref].

Because of the rapid increase in velocity and volume of the result data, traditional relational databases are no longer adequately suitable for such type of data. Especially as they do not scale well to very large sizes (their scaling is vertical; more data means a bigger server) and large data solutions become very expensive using relational databases. NoSql (not only sql) databases like mongoDB become an outstanding candidate for big data storing. Firstly, NoSql databases have been designed to provide good horizontal scalability for read/write database operations distributed, as well as have the ability to replicate and to distribute data, over many servers. These multiple servers can be cheap commodity hardware or could instances, it is a lot more cost effective than vertical scaling of relational databases. Secondly, being divorced from normal table relationships and constraints, as well as ACID (atomicity, consistency, isolation, and durability) of data for increasing the performance, their read/write operations are very faster than the traditional databases. The result storage is used to store the big-volume and high-velocity statistic results from the data processing engine. Hence, we need a distributed data storage. It also has the ability to write data fast in order to avoid increasing the processing time of the data processing engine. So that mongoDb âĂŞ the most popular NoSQl database is selected for this position.

One other reason of choosing the distributed message queue as a component of GPSInsights is to minimize the number of data which loss while recovering from failures. In the case without the message queue component, the transportation data are sent directly to the data processing engine. If a master node of a Spark Streaming cluster dies or "Spout" component of a Storm cluster crashes, they cannot receive any data which arrive during this time. By contrast, if the number of failure node is smaller than the number of nodes keeping the input data (for example, if received data are replicated across two nodes, the queue can tolerate single node failures), the queue always receive the arrival data in spite of the failures of the data processing engine. When recovering, the data processing engine continues pulling the unprocessed data from the queue and then handling them.

However, after successful connecting between different components of GPSInsights, we realized that there were some inherent problems in the system. Firstly, the system could not guarantee that the output data from Spark Streaming were completely sent to MongoDb. This happened when Spark Streaming went wrong, while pushing the result data to the result database. MongoDb in this case just received the incomplete set of the data, but Spark Streaming supposed it completed the task with the current batch and then continued handling the following batch. Hence, that there would be some records which will be lost in the failure case.

Secondly, some records might appear repetitively in a batch due to failures. Because after the received data are replicated persistently on HDFS, the Spark Streaming Receiver informed Kafka about this, so the problem can occur when a part of received data are saved on the reliable filesystem, but Spark Streaming failed before updating for Kafka. Therefore, Spark Streaming supposed it received the data,

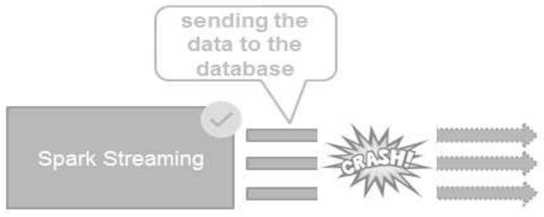

Figure 2: The first problem

but Kafka supposed that the data was not sent successfully. So Kafka would resend the data after the streaming data processing engine recovered from the failure.

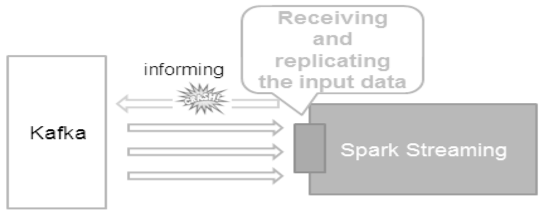

Figure 3: The second problem

The cause of the above problems is that each component of GPSInsights cannot know exactly whether the data are handled fully by the other component or not. To overcome these, only one component needs to maintain a consistent view of what has been processed successful by the system. Therefore, we decided to build a transaction of writing the result data to MongoDB. It means that either all output data from Spark Streaming are logged to MongoDB or the arriving data are reprocessed by the system.

To build this, firstly, we rewrote Spark Streaming's Receiver by using Spark Streaming's "receiverStream" API and Kafka's Simple Consumer API [* ref link] to take advantage of the Kafka's ability of replaying the data from arbitrary offsets. Instead of only handling the latest data when using the old Receiver, with the new one, we can arbitrarily specify any start position of the offset for each partition at the beginning of every batch interval. Besides, we can get the extra information of each record including its offset, id of the partition which it belongs to. Secondly, we created a MongoDb database with three different collections namely "Transactions", "Records", "OffsetRanges". "Transaction" includes documents have three fields: id, timestamp and status. There are two value of status field, "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 will be 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 [* ref link], we can find the offset range of each Kafka's partition in the current batch. When finishing handling this batch, 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. DEMONSTRATION: SCALABLE MAP MATCHING

4.1 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.

time_stamp	car_id	lon	lat	speed

4.2 Algorithm: Road Reverse Geocode Algorithm Using K-D Tree.

4.2.1 *Process OSM raw data.*

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's name, coordinate, ..). Way tag contains one or more node tags that used to define the shape of it.

Before going to the details, vertices, links and segments should be defined clearly. Link is a section of road between intersections. In most digital maps, a real road 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. The road AD in Fig. 4 is formed by 3 intersections (black points), 2 vertices (white points), 2 links (AD, DE) and 3 segments(AB, BC, CD).

Our purpose is to determine all links and their information (name of roads which contain these links). Firstly, we traverse the tag list of OSM, read all Way tags, ignore irrelevant ways (roads that are unfit for transports such as cars, taxis and buses) and then store all associated node ids (we use BTree for this task). If a node occurs once, it is a vertex. Otherwise, it is an intersection. Secondly, we store all Node tags for later uses.

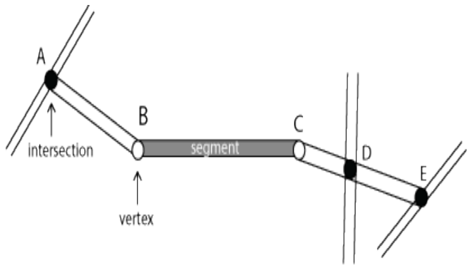

Figure 4: The composition of the road network

4.2.2 *Link matching.*

Because the transport data are collected from GPS tracking device, the input consists of a sequence which contains a time stamp and a geographic coordinate. Our task is to assign a GPS point to a relevant link. Kd-tree is chosen to tackle this problem [8]. 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.

Despite the fact that the standard deviation of GPS data could be quite low in the best case, around 3 meters, it can increase several fold due to tree cover, tunnel and other problems. The limited sampling polling time intervals is the second source affecting the accuracy. There are many methods to solve quite effectively this, including vertex-based and segment-based map matching, map matching using the geometric relationship, map matching using the network topology, the data history and so forth. This paper does not focus on this problem, we only use the vertex-based map matching for the simplicity of the process.

We go through the list of ways in OSM and add some equidistant vertices in segments (illustrated in Fig. **??**) and build a KD-Tree based on those vertices. 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 its nearest vertex as well as a link it is matched to.

4.2.3 *Describe our algorithm.*

In this section, we focus on explaining our proposed algorithm in traffic volume statistic. With this algorithm, we query vehicle object q in a specific time period and a GPS bound from Geomesa aimed at achieving following attributes of q: q's latitude, q's longitude and q's speed. All the information we will store in a data structure list âĂŞ input. After getting input data from Geomesa, now, we review how Spark is used in our algorithm. Firstly, we build up a kdTree with OSM data preprocessed and then broadcast it to the nodes in our system (Line 14). By doing this, we can keep a read-only variable cached on each machine instead of shipping a copy of it with tasks, thereby, providing every node a copy of the kdTree which make up a lot of memory and construction time. Secondly, it is necessary for our algorithm to utilize a collection in-parallel in order to construct a distributed dataset from input data list which can be operated on in paralle. Because the parallel collection can be used many times, we will cache it on memory accelerating our program (Line 15).

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

4.2.4 Data Mapping Phase:

All the elements of the parallelized collection (object q) will be passed on the maptopair method of spark which can operate on in-parallel. As every object q go into the method, the nearestPoint method of copies of kdTree cached on each machine can be utilized aimed at finding what point of a road segment (p) is nearest point of object q and how long distance between q and p (Line 19). Because of the fact that coordination GPS sent from satellites will have some deviations in comparison with the real coordinate, so that we have to choose a threshold distance in order to determine whether object q belong to the road segment including p or not (Line 20 - 27). If the distance is smaller than the threshold distance, it will be much easier for us to conclude that the object q belongs to the road segment and vice versa. As a result, our algorithm can remove the object q that its distance with the road segment is too far from, therefore, we can enhance the accuracy of our algorithm to some extent. Finally, in the Line 29, after every object q is matched to a road segment by road segment's ID, we will group the matched pairs by road segment's ID, hence, we can obtain <key, value> pairs with key being a roadID and value being a iterable of vehicleID and speed.

In the next step in the phase, we will continue to process the <key, value> pairs in-parallel by using the maptopair method of spark. The number of vehicle moving on a road segment also the average speed of the road segment will be calculated in the step (Line 33 - 34).

4.2.5 Data Collecting Phase:

In data mapping phase, statistic numbers of the road segments will be processed in-parallel on the nodes of our system, so that collecting the data perform a decisive role in our algorithm. After the figures are processed, we will use the collect method of spark to convert the parallel collection into a simple list in order to summarize the information and, therefore, we will achieve statistic numbers of traffic volume and average speed on each road segments (Line 38 - 40).

The Pseudocode of Algorithm

5. EXPERIMENTAL EVALUATION

Our system set up on a cluster of HPCC super computer which has one master node and three compute nodes . The configuration of each node include an Intel Xeon E5-2670 Processor (20M Cache, 2.6 GHz, 8 Core), 32 GB DDR3 RAM and FDR 56Gbps Infiniband.

With this configuration, we have conducted several experiments on our data set shown in section 4.1 aimed at appreciating execution time and performance of the architecture. After querying data from Geomesa according year,

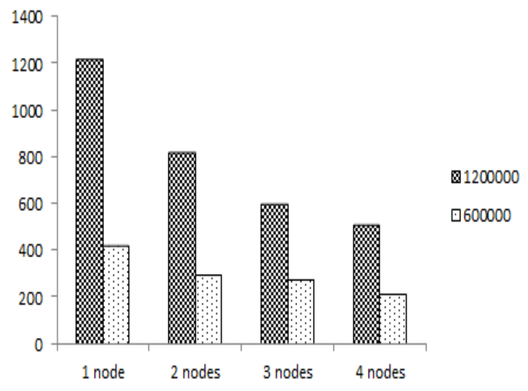

Figure 5: The execution time of the system

month, day of month and hour of day and removing records that its speed attribute is 0, we obtained 1,200,000 records. We compared the running time of the system on the whole records and the half records with various numbers of nodes. The result is illustrated by Figure 6.

It is worth noting that as the figure for nodes was increased, the execution time of the system reduced steadily due to the parallel procedure of spark. While effective indexing strategy of Geomesa enable the architecture to quickly query data from Geomesa, spark help the algorithm to accelerate computation by processing in-parallel. As shown in Figure 6, similarity to 600,000 records, with 4 nodes established, the system take a few more than 500s for 1,200,000 records and less than halving of the running time of the system since we only run on 1 node - a considerable decrease. Besides, we installed a common system running without parallel and kind of indexing strategy like Geomesa such as: using Hash Map with normal database in order to have a comparison with our system. With 1,200,000 records, after 20,000s running, there was no signs of stopping of the system.

6. RELATED WORK

PostGIS [10] is a spatial database extender for PostgreeSQL object-relational database, adds support for geographic objects allowing location queries to run in SQl. Remember that the relational database has the ability to use information from multiple indexes to determine how best to search for the records that satisfy all query criteria, so it is good for multi-dimension data rather than a key-value store. It is undeniable that PostGis completely meets most of the spatial-temporal query and there are a large number of software products that can use PostGIS as a database backend [13]. However, 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. With the reason that the capacity of a PostGIS is only about five hundred millions rows, we have to buy more hardware to meet the demand. However, we will have to cope with many issues such as data management, hardware cost and performance. It is clearly that Accumulo and Geomesa are good for tackle big transport data set.

There are a number of studies on matching GPS observations on a digital map. We can generally classify them into two categories [14] Map-matching algorithms using only geometric relationships between GPS data and a digital map and Map-matching consider not only geometric relationships but also the topology of the road network and history of GPS data.

Map-matching in first category can be classified by Noh and Kim(1998) [9] into the map matching algorithm using the distance of point-to-curve. One using the distance of curve-to-curve and one using the angle of curve-to-curve.

Map-matching using the network topology and the data history match the present point to a link based on the result of matching the previous point. However, when the distance between them is higher than a threshold then this method will use only the geometric relationships to match the GPS points to the nearest link.

Map-matching using the network topology and the data history are not appropriate for our GPS data which is described in the section 4. Neither of those two categories has demonstration with a scalable algorithm building with a distributed processing engine.

7. CONCLUSION AND FUTURE WORK

In this paper, we proposed an efficient framework to manage and process massive amount of GPS vehicle data. With the framework, we can conduct vehicle data statistics efficiently. Thus, we believe that GPSInsights can address an increasing number of the problem classes relating massive GPS vehicle data. We intend to pursue this framework in three directions. Firstly, we plan to use the framework in order to re-imitate routine of a vehicle with a dataset that the period of time between 2 times recording GPS coordinates is significant. Secondly, we research to use this framework for transport state prediction. And finally, the framework also can be used to build up the fastest path finding system.

8. REFERENCES

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

[2] 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.

[3] E. Sivaraman and R. Manickachezian. High performance and fault tolerant distributed file system for big data storage and processing using hadoop. In *Proceedings of the 2014 International Conference on Intelligent Computing Applications*, ICICA '14, pages 32–36, Washington, DC, USA, 2014. IEEE Computer Society.

[4] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: a distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):pp. 4:1–4:26, 2008.

[5] F. Junqueira and B. Reed. *Zookeeper: distributed process coordination.* O'Reilly Media, Inc, 2013.

[6] 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.

[7] 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.

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

[9] Noh and Kim. A comprehensive analysis of map matching algorithms for its. *Hongik Journal of Science and Technology*, 9:pp. 303–313, 1998.

[10] PostGis. http://postgis.net/.

[11] 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.

[12] Sqrrl. http://sqrrl.com/iterators-accumulo/, 2013.

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

[14] 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.

[15] Zookeeper. http://zookeeper.apache.org/.

APPENDIX

A. HEADINGS IN APPENDICES

The rules about hierarchical headings discussed above for the body of the article are different in the appendices. In the **appendix** environment, the command **section** is used to indicate the start of each Appendix, with alphabetic order designation (i.e. the first is A, the second B, etc.) and a title (if you include one). So, if you need hierarchical structure *within* an Appendix, start with **subsection** as the highest level. Here is an outline of the body of this document in Appendix-appropriate form:

A.1 Introduction

A.2 The Body of the Paper

A.2.1 Type Changes and Special Characters

A.2.2 Math Equations

Inline (In-text) Equations

Display Equations

A.2.3 Citations

A.2.4 Tables

A.2.5 Figures

A.2.6 Theorem-like Constructs

A Caveat for the TEX Expert

A.3 Conclusions

A.4 Acknowledgments

A.5 Additional Authors

This section is inserted by LaTeX; you do not insert it. You just add the names and information in the `\additionalauthors` command at the start of the document.

A.6 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references) to create the .bbl file. Insert that .bbl file into the .tex source file and comment out the command `\thebibliography`.

B. MORE HELP FOR THE HARDY

The acm_proc_article-sp document class file itself is chock-full of succinct and helpful comments. If you consider yourself a moderately experienced to expert user of LaTeX, you may find reading it useful but please remember not to change it.