Procedia Computer Science

Volume 80, 2016, Pages 2221–2225

ICCS 2016. The International Conference on Computational Science

Processing High-Volume Geospatial Data: A case of monitoring Heavy Haul Railway operations

Prajwol Sangat[1], Maria Indrawan-Santiago[1], David Taniar[1], Beng Oh[2], and Paul Reichl[2]

[1] Faculty of Information Technology, Monash University, Australia
[2] Institute of Railway Technology, Monash University, Australia
{prajwol.sangat, maria.indrawan, david.taniar, beng.oh, paul.reichl}@monash.edu

Abstract

Sensor technology such as GPS can be used in the mapping of transportation networks (e.g., road, rail). However, GPS suffers from errors in positional accuracy due to factors such as signal arrival time. In railway systems, positional accuracy is of utmost importance to identify state of track and wagons for safety and maintenance. Along with GPS, the numerous lightweight sensors installed in each wagon produce a high-velocity geospatial data that needs to be processed continuously and the traditional data processing and storage applications can not handle it. We propose efficient algorithms and a suitable data structure to achieve rapid and accurate location mappings. Our large-scale evaluation demonstrates that the system is accurate and capable of real-time performance.

Keywords: Geospatial Data, Big Data, Batch Data Processing

1 Introduction

Global Positioning System (GPS) are incorporated in many small devices such as mobile phones and sensors. They provide location information which can be used to track the path of an object along a time line. In the mining industry, the heavy haul trains carry a massive load and perform round trips from a loading point to a port. so, the ravages to the tracks and wagons are inevitable. Train tracks and wagons need regular monitoring and maintenance. Thus, heavy haul railways need GPS and other sensors to monitor performance and identify defects [2].

Processing the sensor readings collected for monitoring railway safety poses several challenges. Firstly, the readings from GPS are inaccurate because of which it is possible that the current position of a particular wagon is read to be in a wrong track segment or not on any of the segments due to a mismatch in the longitude and latitude values compared.

Secondly, the data generated has high velocity and volume because of numerous cheap and lightweight sensors installed in each wagon. These sensors collect data ceaselessly during the round trips of trains. However, it is not guaranteed that the sensors will work perfectly all the

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.385

time causing data to have veracity. The main problem is the high velocity of incoming data that needs to be processed on the fly and storing of data with the uncertain structure before it can be used for monitoring and analysis purposes. The traditional data processing and storage applications can not deal with such velocity, veracity and volume.

In this paper, we describe our proposed algorithms and system to perform location matching for heavy-haul railway monitoring. The algorithms and their implementation are based on distributed processing framework - Apache Spark [9]. The paper is organised as follows: Section 2 describes the design and implementation of algorithms. Section 3 describes the experimental setup and Section 4 describes the lessons learnt during the development of the system.

2 Geospatial Data Processing for Railway Monitoring

Geospatial data hold information about geography and can have extra information, such as track data. GPS makes error corrections for receiver clock error, but there are residual errors such as geometric dilution of precision and signal arrival. So, **"Geo-Hashing"** is used to create a new lookup table to improve the precision of the geographic location of train cars.

Geohashes offer properties like arbitrary precision and the possibility of gradually removing characters from the end of the code to reduce its size (and gradually lose precision). Reducing the number of bits from geohash produces more output as there can be a number of points in the same bucket [3]. In such case, **"Haversine Formula"** is used to find the closest distance between two points using their latitudes and longitudes [6]. In section 2.1, we present the proposed design and implementation of algorithms used in geospatial data processing system. In section 2.2, we discuss the system architecture and the data storage.

2.1 Design and Implementation

"Geo-Hashing" and **"Haversine Formula"** have been used for the implementation of data processing system. The idea is to calculate the geohash with 8-bit precision for each record and insert it into `Map` data structure with geohash as the *"key"* and incoming spatial data itself as the *"value"*. With the precision of 8-bits, it is most likely to have multiple incoming records with the same *"key"*. Therefore, the *"value"* in the key value pair of `Map` data structure is a *"list"* holding the incoming spatial data with same keys. With this approach, the cost of geohash lookup is $O(1)$. Once the list is retrieved using geohash, haversine formula can be used to find the closest distance for the input spatial data. The cost of lookup in the list is $O(n)$, where n is the number of items in the list. The number of items in the list is usually significantly less (not more than 10 to 15 records) therefore the time taken for sequential lookup is minor.

2.1.1 Geo lookup Map

Algorithm 1 constructs a `TreeMap` to store the geohash and corresponding list of locations. The `TreeMap` is sorted and ordered based on the key [4]. `TreeMap` is stored in memory and looked up to get the matching geohash when the actual mapping is done. Geo lookup Map construction algorithm has been implemented in Scala (about 200 lines in total). Its memory consumption is directly proportional to the number of records in the input static dataset.

`TreeMap` is not the only way to store key value pairs. Many other `Map` data structures, such as `HashMap`, `LinkedHashMap` can also be used [4]. In the future, we plan to investigate other data structures to determine if they are better suited for particular data-set and problem.

2.1.2 Search Closest Location

Algorithm 2 illustrates the use of haversine to find the closest location. After retrieving the list with the same geohash from the `TreeMap`, a simple mathematical formula of comparing the distance and storing the location with the least distance can be used to find the closest location.

The output of the algorithm is the closest location (CL) from the input geospatial record and is used to append additional information attached to CL (i.e. TrackName, TrackID etc.) into the input geospatial record before it is dumped into MongoDB.

Algorithm 1: Construct Geo LookUp Map

Input: L: Lookup Dataset
Output: T: A Geo LookUp TreeMap

```
Initialize T as an empty TreeMap
for l ∈ L do
    Split individual items in l which includes Lat and Lon
    Get the geocode using Lat and Lon
    if geocode NOT IN T then
        locationlist ← l
        T ← {geocode, locationlist}
    else
        locationlist ← T[geocode]
        locationlist ← l
    return T
```

Algorithm 2: Search Closest Location

Input: T: A Geo LookUp TreeMap,
Lat: Latitude, Lon: Longitude,
S: Stream of Records
Output: CL: Closest Location

```
Initialize refDistance ← 1000
for s ∈ S do
    Get the geocode using Lat and Lon
    if geocode IN T then
        locationlist ← T[geocode]
        for location ∈ locationlist do
            distance ← get distance using Haversine Formula
            if distance < refDistance then
                refDistance ← distance
                CL ← location
    return CL
```

2.2 System Architecture

With the use of distributed data processing technologies such as Apache Spark [9], it is easier to partition data and process it in parallel. The *shared-everything* (i.e. shared memory and shared disk) *architecture* is followed and data is assigned logically to each processor. The data is *horizontally partitioned* in which each processor holds a partial number of complete records of input data [8]. The lookup operation executes in parallel for each partition and requires all processors to participate. With this technique, full utilisation of server resources is possible and is crucial to performance.

The geospatial data processing system needs to support tens of thousands of records every second and handle veracity of data. MongoDB is flexible and supports dynamic structure [5]. MongoDB provides lower execution times than SQL databases in all four basic CRUD operations [1, 7]. MongoDB is used because the application is data intensive and needs to store a large amount of schemaless data and efficiently query them in the future.

3 Evaluation

In this section, we present our experimental setup and illustrate the effects of various settings (e.g. data size) on the two performance indicators of our system: efficiency and accuracy.

3.1 Details on Data Processing Technology and Host System

Version 1.4.0 of Apache Spark was used for the Spark implementation. The algorithms were implemented in Scala 2.10.4. The testing environment on which algorithms were tested is the National eResearch Collaboration Tools and Resources (NeCTAR)[1] cloud computing environment. The instance has 16 dedicated cores, 64 GB of memory, 459 GB HDD and runs 64 bit Ubuntu GNU/Linux 14.04.2 LTS.

3.2 Evaluation Methodology

To evaluate the system, the key metric to consider is "time" as we are focused on improving the performance of the system.

1. **TreeMap Creation Time:** An iterative creation of `TreeMap` is performed to note the time taken for its creation. We have used the static data set with 200K records.
2. **Time to Process Geospatial Data:** The actual number of records expected in the real system is approximately 36K per second. We created chunks of 5K, 10K, 15K, till 40K and passed it to processing component to record the processing time. Also, we created random chunks of 40K records and tested the component for processing of incoming data with the continuous input of 40K different records every second.

3.3 Results

1. **Accuracy:** Table 1 shows the percentage accuracy in matching track information using 250 million historical records which included trip information collected by sensors in 5 different trains operating in Pilbara region. We observe in Table 1 that the average percentage match is around 98.5%. The errors found are mostly due to edge predictions. Edge predictions have a very high standard deviation, indicating that such predictions are not often correct which is a case of 1 in 1,000.

No. of Records (Millions)	No. of Errors (Thousands)	Accuracy (%)
100	1586	98.54
150	2420	98.39
200	3070	98.47
250	3720	98.51

Table 1: Percentage accuracy of mapping track information

2. **TreeMap Creation :** For this test, 50 `TreeMap` were created from the same 200K static dataset iteratively. For each iteration, it took on average 6 seconds for the creation of `TreeMap`. In the actual system, the process runs as a pre-processing task before the system starts and the performance improvement it provides is remarkably better, helping us ignore the one-time setup of $\approx$ 6 seconds.
3. **Processing Time Vs. Number of Records :** Figure 1 shows the linear increase in the mapping time with the increase in the number of records. As can be seen from Figure 1, 5K records require nearly 440 milliseconds to process, 10K records require nearly 550 milliseconds and the pattern continue with the small increment in time ($\approx$ 100 milliseconds) with the increment in the number of records by 5K. The evaluation is ceased at 40K records which is processed under a second ($\approx$ 0.8 to 0.9 seconds) as the maximum records per second expected from the actual system is 36K.

[1] https://www.nectar.org.au/about-nectar

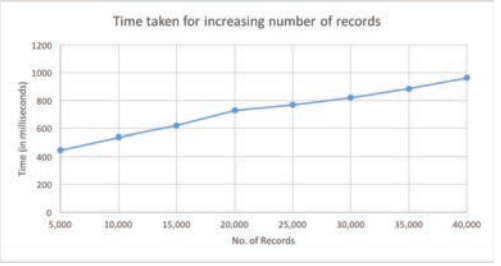

Figure 1: Processing time for different number of records

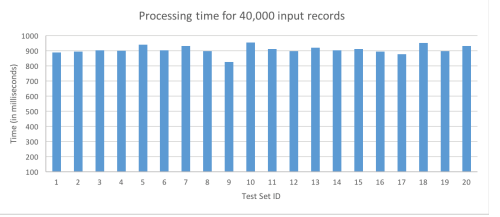

Figure 2: Mapping time for 40K records (20 test set)

Also, the system is iteratively tested (20 test set) for 40K inputs. Figure 2 depicts that the processing time remains consistent between $\approx$ 0.8 to 0.9 seconds i.e. the performance of the system is consistent with random inputs till 40K records.

4 Conclusion

We have described the design and testing of geospatial data processing system that provides fast and accurate track information mapping and can run on commodity server hardware. This process has made us re-learn many of the systems building fundamentals such as reducing the data size through aggregation and smart filtering is essential in real time performance on commodity hardware, simple algorithms are frequently good enough and make it much easier to debug problems and partitioning of large amount of data to achieve proper CPU load balancing is vital and determines the run time efficiency of the system. We hope these observations will prove useful in reducing the system-building time for other researchers and developers.

References

[1] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[2] Michael Darby, Eugenio Alvarez, John McLeod, Graham Tew, Gregory Crew, et al. Track condition monitoring: the next generation. In *Proceedings of 9th International Heavy Haul Association Conference*, volume 1, pages 1–1, 2005.

[3] Gustavo Niemeyer. Tips & Tricks - geohash.org. Accessed: 2015-11-30.

[4] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[5] Chodorow Kristina and Dirolf Michael. Mongodb: The definitive guide, 2010.

[6] CC Robusto. The cosine-haversine formula. *American Mathematical Monthly*, pages 38–40, 1957.

[7] Michael Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11, 2010.

[8] David Taniar, Clement HC Leung, Wenny Rahayu, and Sushant Goel. *High performance parallel database processing and grid databases*, volume 67. John Wiley & Sons, 2008.

[9] 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*, volume 10, page 10, 2010.