Distributed Moving Objects Database Based on Key–Value Stores

Hong Van Le
Supervised by Atsuhiro Takasu, takasu@nii.ac.jp
SOKENDAI (The Graduate University for Advanced Studies), Japan
l-van@nii.ac.jp

ABSTRACT

Moving objects databases (MOD) have been studied intensively and extensively in the database field. Recently, the emerging Big Data trend, which refers to a collection of large, complex, and rapidly growing geographical data collected from sensors and GPS-enabled devices, has posed new requirements for MOD: the ability to manage massive volume of data, the support for low-latency spatio-temporal queries and the need for high insertion throughput rate. Although key–value stores handle large-scale data efficiently, they are not equipped with effective functions for supporting spatio-temporal data. In this project, we aim to build a distributed MOD which fulfills all these requirements. We focus on the design of an efficient distributed spatio-temporal index that can support indexing and querying moving object data in key–value stores.

1. INTRODUCTION

Recent rapid improvements in positioning technologies such as satellites, the Global Positioning System (GPS), sensors, and wireless networks has resulted in an explosion of data generated by moving objects. For instance, traffic management systems in busy cities such as Tokyo have been collecting large numbers of location updates from probe cars such as taxis, buses and GPS-enabled devices at a rate of multiple updates per minute from each vehicle. The collected data often include location data (longitude and latitude) and time data (a timestamp).

Many systems, in both scientific research and daily life, have taken advantage of the data collected from moving objects. For example, intelligent transportation systems are exploiting the massive volume of sensor data from probe cars and GPS-enabled devices to provide efficient route planning and traffic balancing based on the current traffic situation. People can take advantage of these systems by continuously sending data about their current location and time, then receiving relevant real-time analyses of their traffic situation. Such systems require a MOD; that can store massive volumes of data, support high insertion throughputs, and respond to queries in real-time.

There are three main challenges for a MOD to be useful for these systems. *The Volume Challenge.* The database should have high scalability, fault-tolerance and availability while dealing with large volumes of collected data. *The High Computational Complexity Challenge.* Many queries will involve geometric computations, such as very expensive logical operations on spatial relationships, that are continuously changing. *The Real-time Processing Challenge.* Because many agents keep registering their location updates continuously, the database must have high insertion throughput to handle the volume of data. It must also guarantee satisfactory performance on queries, but as the dataset become bigger, query time can increase dramatically.

Some systems have been proposed, such as SpatialHadoop [6] and Hadoop-GIS [1], that are based on MapReduce, a parallel, distributed and scalable framework for processing large volumes of data. These systems support high-performance spatial queries, but so far have not provided support for temporal constraints with spatial data. Moreover, such systems are suitable for batch processing; they still have a high latency compared with the requirements of a real-time system. Key–value stores (KVS) such as HBase[1], with their scalability, fault tolerance, availability and random real-time read/write capability, have shown promise. However, they do not have native support for spatial and spatio-temporal queries.

Spatial support has been extended to KVS. For instance, MD-HBase [11] layers a multidimensional index over the KVS by using Z-order and multidimensional index structures. However, it does not support spatial queries concerned with evolution, in which spatial relationships change over time. GeoMesa [7] introduces a spatio-temporal index structure based on Geohash[2] on top of Apache Accumulo[3]. It interleaves Geohash and timestamps into the design index and achieves acceptable results when storing data using 35-bit Geohash strings. Nonetheless, the performance of its proposed spatio-temporal index depends significantly on the number of Geohash characters in the rowkey and on the resolution level; hence, further study of their impact on queries and dataset is required.

The goal of our work is to build a complete MOD based on KVS that overcomes the above challenges. We first proposed a novel two-tier index structure to handle spatial data

[1]https://hbase.apache.org
[2]http://geohash.org
[3]http://accumulo.apache.org

Proceedings of the VLDB 2016 PhD Workshop, September 9, 2016. New Delhi, India.

in HBase. Our experimental results indicate that queries using the proposed index outperformed queries with other indexes and queries in a MapReduce-based system. Then we indexed moving objects data by presenting a lightweight spatio-temporal index structure based on STCode [9], an encoding algorithm that treats time in the same way as spatial information. However, time is not just another dimension but a special one beside the two spatial dimensions, so it is necessary to explore new methods for incorporating temporal information into spatial indexes. We discuss this further in Section 3 and briefly describe our research plan in Section 4.

2. KEY–VALUE STORES

In our project, we build the database based on the architecture of KVS. This is modeled after Google's BigTable [3] such as Cassandra, Accumulo, HBase. These KVS share some important characteristics inherited from BigTable. In this section, we describe these characteristics of HBase, as we use it in our experiments.

HBase is a distributed scalable database that takes advantage of the HDFS. Tables in HBase include rows and columns like other databases, but they can scale to large numbers of rows and columns. To store such large tables in a distributed cluster, tables are split into a number of smaller chunks called regions, which are stored in servers called RegionServers.

HBase is also a KVS since its data model is organized as a KVS. Within a table, data are stored as a sorted list of key-value pairs according to rows that are uniquely identified by their rowkeys, which therefore play an important role when searching or scanning. Rowkey design is one of the most important aspects of HBase schema as well as other KVS.

The physical data model in HBase is column-oriented, making it a column-oriented database. Rows in HBase are composed of columns and columns are grouped into column families. Columns in a family are stored together in a low-level storage file called an HFile. The column family forms the basic unit of physical storage to which HBase features such as compression can be applied. Hence, proper design of column families is also essential when storing and processing data on HBase.

3. COMPLETED AND ON-GOING WORK

3.1 Spatial Index

3.1.1 *Two-tier index structure*

Data stored in HBase are accessed by a single rowkey. However, spatial data are represented by two coordinates (longitude and latitude), which are equally important in defining a location. Geohash provides a solution to transform longitude/latitude coordinates into unique codes. Figure 1 shows how to generate a Geohash code for a spatial point. Geohash recursively performs binary partitioning to divide the range into two equal parts, and then assign bit 0 if the location is in the left part and bit 1 if it is the right part, respectively. Then it interleaves bits from the two dimensions and uses base 32 to encode the bit sequence into a hash code.

There are some advantages of using Geohash to store spatial data into KVS. Points that share the same Geohash

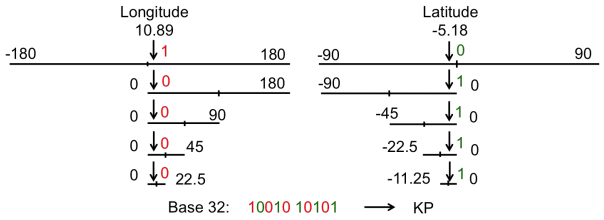

Figure 1: Geohash generation

prefix will be close to each other geographically. If we store objects in lexicographical order of row keys in the KVS, objects that are close to each other spatially will also be close to each other in the database and we can scan relevant objects efficiently by using prefix filters. However, using Geohash alone does not guarantee efficient spatial query processing. For instance, a prefix filter may give insufficient results when finding k nearest neighbors of a point that is near the rectangle border of the prefix. Geohash uses Z-order to order the rectangles, as shown in Figure 2, so when the system scans all points ranging from rectangles with Geohash *1* to *4*, it will scan the unrelated rectangles *2* and *3*. To obtain accurate query results and prune unnecessary scanning, we utilized the R-Tree, a multidimensional index structure for geographical data, as the secondary index tier.

Figure 2: Z-order of 1-character Geohash codes

To bridge the two tiers Geohash and R-Tree, we propose a novel data structure, the binary Geohash rectangle-partition tree (BGRP Tree) [10]. We first partition regions into rectangles using the *longest common prefix*. Because overlap between these rectangles would lead to redundant scans if we insert them directly into the R-Tree, we use the BGRP Tree for further partitioning of rectangles into subrectangles, until there is no overlap between them. Finally, we insert all subrectangles into the R-Tree. When processing spatial queries, the system finds the rectangles in the R-Tree that may contain query results. Then scan process is conducted on the rectangles found, allowing us to prune the scanning on unrelated regions.

3.1.2 *Experimental results*

We built a cluster with 64 nodes. Each node had a virtual core, 8 GB memory and a 64 GB hard drive. The operating system for the nodes was CentOS 7.0 (64-bit). We set up one HMaster, 60 Region Servers and three Zookeeper Quorums using Apache HBase 0.98.7 with Apache Hadoop 2.4.1 and Zookeeper 3.4.6. Replication was set to two. To conduct queries on SpatialHadoop, we installed SpatialHadoop v2.1,

which shipped with Apache Hadoop 1.2.1, and configured one master and 64 slaves.

We evaluated the insert performance using Yahoo! Cloud System Benchmark. The number of workloads was varied from one to 64. Because real spatial data are often skewed, we chose a Zipfian distribution to generate longitude and latitude of each input point. We observed that the system could sustain a peak throughput of 600 K inserts per second when there were eight workloads generating data. However, when the number of workloads was increased to 32 or 64, we observed a drop in performance. This was because of the cost of splitting regions when a region becomes a hot spot. Moving objects data are naturally skewed but as shown in Figure 2, Geohash divides the space into equal buckets, so some prefixes in dense areas (e.g., *u*, *s*, *w*) are used more frequently than the prefixes in sparse areas (e.g., *0*, *1*, *2*, *3*), and regions of the prefixes in dense area can easily become hot spots.

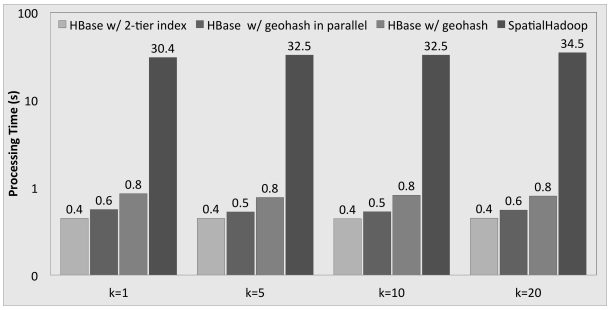

Figure 3: KNN queries with T-Drive dataset

We executed k nearest neighbours (kNN) queries using the two real datasets, T-Drive [13] and OpenStreetMap (OSM)[4]. Figure 3 shows the performance of kNN queries for the T-Drive dataset, which is more than 700 MB with 17,762,390 records. With both datasets, we observed that parallel kNN using a two-tier index outperformed all other kNN queries with Geohash only. Note that our index design operates about 60 to 90 times faster than SpatialHadoop. This is because HBase does not require the startup, cleanup, shuffling and sorting tasks of MapReduce. Another reason is that we store kNN procedures in every region server beforehand, thereby needing only to invoke that procedure locally on each server. In contrast, MapReduce sends a procedure to slave servers for every query, thereby requiring more time for network communication.

3.2 Spatio-Temporal Index

A spatial index is important in indexing moving objects, but we cannot apply it directly to create a moving-objects index because of the evolution of objects over time. In our current approach, temporal information is simply treated as an extra dimension on top of the spatial hash. For instance, STCode [9] is an algorithm that encodes spatial information (longitude, latitude), and temporal information into unique codes. Latitude is $\lambda \in (0, 180)$ and longitude is $\varphi \in (0, 360)$. Temporal information is represented in minutes within 1 year $m \in (0, 527{,}040)$. The minute values ($527{,}040 = 366 \times 24 \times 60$) cover the whole year, even in a leap year. STCode is generated by applying the rule used with Geohash to divide the range of each dimension into two equal parts and interleave bits from the three dimensions, then finally using base 64 instead of base 32 to encode the bit sequence. Figure 4 shows an example of generating an STCode of two characters for a point.

[4]http://www.openstreetmap.org

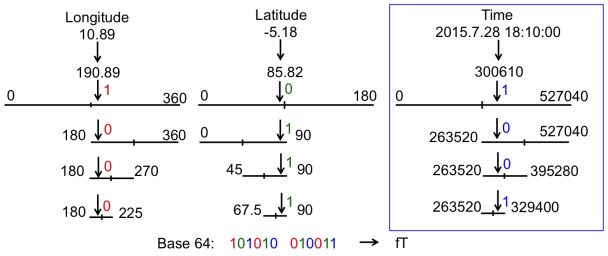

Figure 4: STCode generation

Because KVS keeps rowkeys in lexicographical order, we can exploit STCode as a rowkey on HBase. Since STCode cannot encode the year information, we achieve the same STCode for points that are in the same place and time but in different years. To handle this limitation, we stored the year information as a column family in HBase. The reason we chose a column family is that it is the secondary information to identify a row beside the rowkey, and the basic unit of physical storage in HBase. By using year information as a column family, we can distinguish data for different years and store or compress all data for a year in the same or nearby physical storage, thus improve the performance when searching for close points in the database.

4. RESEARCH PLAN

In this section, we describe the key future directions to meet the requirements of a MOD and to overcome some limitations of our previous work as discussed in Section 3.

Space-filling curves: Transforming from two-dimensional spatial data into one-dimensional data is essential to store moving objects data into KVS. In previous work, we have used Z-order because of its simplicity for both encoding and decoding, so it could support high insertion throughput. However, the quality measures for space-filling curves in [8] indicate that Z-order performs worst. Therefore, we would like to study the performance of other space-filling curves including Hilbert order, $\beta\Omega$ order, and AR^2W^2 order in the context of indexing spatial data in KVS, and analyze the relationship between quality measures of the curves and query performance in databases.

Indexing current and near future movement: Temporal information has a special feature—monotonicity. For example, if the evolution of an object is represented by a set of records (o_i, s_i, t_i) where s_i is the location of object o_i at the timestamp t_i, for each two consecutive records of an object o_i (o_i, s_i, t_i) and (o_i, s_{i+1}, t_{i+1}), we always have $t_{i+1} > t_i$. Because of this feature, some moving objects data could be considered as obsolete if their timestamps are smaller than a predefined threshold. To handle these obsolete entries, pack and purge operations could be employed to save index and storage space. Purging obsolete entries and reorganizing after purge operations must be considered while designing index structures. Separately maintaining temporal and spatial information influences purge operations less than other queries, which are often related to recent data.

Therefore, instead of indexing temporal information as another dimension, which occurs in STCode, incorporating it as a special dimension is a potential and challenging approach in our research.

Density based partition: While Z-order as well as some other space-filling curves divide the surface into equal buckets, moving objects data are naturally skewed. This leads to some problems in handling spatial queries. First, as mentioned in Section 3.1, the load imbalance of data insertion into distributed storage could reduce the insertion throughput. Second, it also causes many false positive scans if the scanning bucket is bigger than the query range. On the other hand, if we divide the query range into a large number of small buckets to scan, the I/O increment would cause significant performance deterioration. To overcome these problems, we should partition the space according to the density of the data.

5. RELATED WORK

Storing and querying geospatial data have been supported by some NoSQL databases such as document-oriented databases (CouchDB, MongoDB) or network-oriented (Neo4j). [4] provides an intensive survey of NoSQL Geographic Databases. Although MongoDB supported 2D and 2Dsphere indexes for spatial data, it deals only with point vectors and its input and output are limited to GeoJSON. GeoCouch, the spatial extension of CouchDB, cannot filter based on both spatial and nonspatial criteria in a query. The spatial library of Neo4j, Neo4j Spatial, supports many types of spatial data and almost fully supports spatial functions but still requires several improvements to index and query temporal information with spatial information.

With the development of large main memories, many researchers have proposed main-memory indexes to deal with the challenge of trading off between high update rates and low-latency location-based queries. TwinGrid [12] and MOVIES [5] maintain two separate index structures, the update and query indexes. The update index stores all arriving updates during a time period. The query index is a read-only index to answer queries. TwinGrid replaces the query index by the update index periodically while MOVIES accumulates the update index and rebuild the query index from the accumulated update index. These systems can support high update rates but suffer from the staleness of query results. ToSS-it [2] used Voronoi diagrams to index spatial data and a distribute-first build-later approach to distribute data to servers. It provides high query throughput and scalability but when data are skewed, it takes time to update the index.

6. CONCLUSION

This paper discusses the challenges of building MOD that can support large volumes of data and provide both low-latency queries and high insertion throughput. We have proposed an index structure to support spatial data on KVS and observed improved query performance compared with other index structures and with the state-of-the-art framework based on MapReduce. We are in the early stages of integrating temporal information into spatial index structures to support moving objects data. We described the problems left to solve and the approaches and plans to achieve the goal of this PhD project.

7. REFERENCES

[1] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020, 2013.

[2] A. Akdogan, C. Shahabi, and U. Demiryurek. Toss-it: A cloud-based throwaway spatial index structure for dynamic location data. In *Mobile Data Management (MDM), 2014 IEEE 15th International Conference on*, volume 1, pages 249–258. IEEE, 2014.

[3] 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 Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[4] C. de Souza Baptista, C. E. S. Pires, D. F. B. Leite, and M. G. de Oliveiraa. Nosql geographic databases: an overview. *Geographical Information Systems: Trends and Technologies*, 73, 2014.

[5] J. Dittrich, L. Blunschi, and M. A. V. Salles. Movies: indexing moving objects by shooting index images. *GeoInformatica*, 15(4):727–767, 2011.

[6] A. Eldawy and M. F. Mokbel. A demonstration of spatialhadoop: an efficient mapreduce framework for spatial data. *Proceedings of the VLDB Endowment*, 6(12):1230–1233, 2013.

[7] A. Fox, C. Eichelberger, J. Hughes, and S. Lyon. Spatio-temporal indexing in non-relational distributed databases. In *Big Data, 2013 IEEE International Conference on*, pages 291–299. IEEE, 2013.

[8] H. Haverkort and F. van Walderveen. Locality and bounding-box quality of two-dimensional space-filling curves. *Computational Geometry*, 43(2):131–147, 2010.

[9] J. Ježek and I. Kolingerová. Stcode: The text encoding algorithm for latitude/longitude/time. In *Connecting a Digital Europe Through Location and Place*, pages 163–177. Springer, 2014.

[10] B. Le Hong Van and A. Takasu. An efficient distributed index for geospatial databases. In *Database and Expert Systems Applications: 26th International Conference, DEXA 2015, Valencia, Spain, September 1-4, 2015, Proceedings*, volume 9261, page 28. Springer, 2015.

[11] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, volume 1, pages 7–16. IEEE, 2011.

[12] D. Šidlauskas, K. A. Ross, C. S. Jensen, and S. Šaltenis. Thread-level parallel indexing of update intensive moving-object workloads. In *Advances in Spatial and Temporal Databases*, pages 186–204. Springer, 2011.

[13] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems*, pages 99–108. ACM, 2010.