CloST: A Hadoop-based Storage System for Big Spatio-Temporal Data Analytics

Haoyu Tan, Wuman Luo, Lionel M. Ni
Department of Computer Science and Engineering
HKUST Fok Ying Tung Graduate School
Hong Kong University of Science and Technology, China
{hytan, luowuman, ni}@cse.ust.hk

ABSTRACT

During the past decade, various GPS-equipped devices have generated a tremendous amount of data with time and location information, which we refer to as *big spatio-temporal data*. In this paper, we present the design and implementation of CloST, a scalable big spatio-temporal data storage system to support data analytics using Hadoop. The main objective of CloST is to avoid scan the whole dataset when a spatio-temporal range is given. To this end, we propose a novel data model which has special treatments on three core attributes including an object id, a location and a time. Based on this data model, CloST hierarchically partitions data using all core attributes which enables efficient parallel processing of spatio-temporal range scans. According to the data characteristics, we devise a compact storage structure which reduces the storage size by an order of magnitude. In addition, we proposes scalable bulk loading algorithms capable of incrementally adding new data into the system. We conduct our experiments using a very large GPS log dataset and the results show that CloST has fast data loading speed, desirable scalability in query processing, as well as high data compression ratio.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Systems and Software

Keywords

Storage System, Big Data, Spatio-Temporal Data

1. INTRODUCTION

In recent years, the growth of mobile devices capable of reporting their locations has led to tremendous interests in both research and industrial communities. Lately, we have been focusing on mining a very large GPS log dataset which contains spatio-temporal and relevant informations including location, time, speed, orientation and vacant status collected from around 17,000 GPS-equipped taxis in Shanghai and Shenzhen. The data have been accumulated for over 2 years. For now, the size of the dataset is about 200GB in uncompressed text format, containing around 2 billion records, and is increasing at a rate of 1–2GB/day. As the data size have posed great challenges in efficient storing and processing of them, we refer to such data as *big spatio-temporal data*.

In face of such large datasets, we use Hadoop [1], an open-source implementation of MapReduce [2] and related components, to perform spatio-temporal data analytics. We choose Hadoop over RDBMS because it provides us with low-level application programming interface (API) which is more flexible for implementing complex data mining algorithms than structured query language (SQL). Based on our analysis of query workloads over the data, we find out that almost all queries have range predicates on location and time, which we refer to as *spatio-temporal range queries*, or simply *range queries*. After examining the number of taxies involved in the query results, we find out that most range queries are either restricted to only one taxi or related to all taxies. Accordingly, we refer to these queries as *single-object query* and *all-object query*, respectively.

However, using Hadoop alone is not quite suitable for spatio-temporal data processing because the underlying storage systems HDFS and HBase do not support efficient multi-dimensional range scan. We have investigated other storage systems that are dedicatedly designed for spatio-temporal data such as TrajStore [3] and PIST [4]. They assume a non-distributed environment makes them undesirable for very large spatio-temporal datasets. Therefore, there is a notable gap between big data analytics and spatio-temporal data storage. It motivates us to design and implement an efficient and scalable storage system for big spatio-temporal data analytics.

In this paper, we present the design and implemtation of CloST, a storage system based on Hadoop to for big spatio-temporal data analytics. Our main contributions are as follows:

- We propose a new data model capable of optimizing the system implementation for spatio-temporal attributes; yet it is sufficiently simple to apply to very large datasets. To the best of our knowledge, CloST is the first system using this data model.
- We describe a three-level hierarchical partitioning approach to enable efficient parallel processing of all-object queries with any possible ranges, i.e, the effectiveness of the data partitioning does not rely on

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'12, October 29–November 2, 2012, Maui, HI, USA.
Copyright 2012 ACM 978-1-4503-1156-4/12/10 ...$10.00.

whether the spatial predicate is more selective than the temporal predicate or on the contrary.

- We describe a space-efficient file format to actual store data on HDFS. In such files, records from the same object are grouped together and then stored by column. Data are compressed twice at column-level and group-level respectively. Comparing to text-formatted raw data files, CloST can reduce the storage size by an order of magnitude.

- We describe parallel algorithms using MapReduce to perform all-object queries, initial data loading and incremental data loading. The experiment results show that these algorithms are scalable and efficient when handling very large datasets.

The rest of the paper is organized as follows. Section 2 presents the background of HDFS and MapReduce. Section 3 presents the design and implementation of CloST in detail. Section 4 presents the performance evaluation results. Section 5 presents an overview of related systems and Section 6 concludes the paper.

2. HADOOP BACKGROUND

MapReduce [2] is a simplified data processing model originally proposed by Google for data-intensive applications. Because of its simplicity, this programming model can be easily realized in parallel and most importantly, it allows elegant handling of failures. During the last decade, MapReduce has been widely used in both academia and industry for big data analytics. Usually, both the input and the output of MapReduce are stored on a distributed file system optimized for sequential access such as the Google File System (GFS) [5]. The Apache Hadoop [1] project provides an open-source implementation of MapReduce along with the Hadoop Distributed File System (HDFS) which is similiar to GFS. Due to the page constraint, we omit the detailed description of MapReduce and HDFS. For more technical details, interested readers can refer to the Hadoop project website [1] and the original papers [2][5]

3. THE DESIGN AND IMPLEMENTATION OF CloST

In CloST, all data and metadata are stored on HDFS as regular files with a predefined directory structure and CloST provides a number of components to manage them. As is shown in Figure 1, the building blocks of CloST include *metadata manager*, *data loader*, *storage optimizer* and *query processor*. We will present the details of each building block in this section.

3.1 Data Model

To support efficient spatio-temporal range scan, we use a new data model in CloST. This data model is targeted at very large spatio-temporal datasets which shares similar characteristics with the GPS log data. It stores spatio-temporal data in *tables*. A table has the schema in the form of ($\texttt{Oid}, \texttt{Loc}, \texttt{Time}, A_1, \ldots, A_n$). The first three attributes are referred to as *core attributes* where `Oid` is an object id, `Loc` is a spatial point, and `Time` is a timestamp. The most important feature of the CloST data model is that core attributes are compulsory and predefined for any table. As such, storage systems based on this data model can particularly optimize for the core attributes. In oppose to core attributes, A_1 through A_n are defined by users and are referred to as *common attributes*.

3.2 Storage Structures

The following describes how data are partitioned, stored and indexed in CloST. We require the storage structures can deal with both single-object queries and all-object queries. To be clear, we use $Q(I, S, T)$ and $Q(S, T)$ to indicate a single-object query and an all-object query respectively, where S denotes a spatial range, T denotes a temporal range, and I denotes an object id.

3.2.1 Hierarchical Partitioning

To enable efficiently processing of both Q_1 and Q_2, CloST partitions data hierarchically using all core attributes. As depicted in Figure 2, data are first divided into a number of level-1 partitions according to hash values of `Oid` and coarse ranges of `Time`. Then, each level-1 partition is divided into a number of level-2 partitions according to a spatial index on `Loc`. Finally, each level-2 partition is further divided into a sequence of level-3 partitions according to finer ranges of `Time`. Level-3 partitions contain actual records and higher level partitions serve as indexes for level-3 partitions. All the records and indexes are stored as regular files on HDFS. In the following, we refer to a level-1 partition as a *bucket*, a level-2 partition as a *region*, a level-3 partition as a *block* and the corresponding file as a *block file*.

Level-1 partitioning roughly breaks a very large dataset into relatively small buckets. As a result, each bucket can independently perform query processing, data loading and storage optimization. Besides, in context of parallel processing, the number of buckets is the minimum degree of parallelism when answering an all-object query. Therefore, we can always parallelize an all-object query even when the range is very small. The essence of level-2 and level-3 partitioning is *first spatial partitioning then temporal partitioning*. We use a quadtree to divide a bucket into regions and then use 1-D range partitioning to divide a region into blocks.

3.2.2 Block File Layout

Motivated by RCFile [6], we devise an efficient and compact storage layout to actually store records. Figure 3 shows an example of the internal structures of a block file. We group records by `Oid` and store each group in a file section. An in-block index is placed at the beginning of the file to map an `Oid` to the corresponding section position. For each section, records are first sorted by `Time` and then organized in a column-store fashion, i.e., values from the same attribute are stored continuously. To save space, we do not store `Oid` in sections because we can infer it from the in-block index.

To improve the storage efficiency, CloST compresses data first at column-level and then at section-level. For column-level compression, numeric values are encoded by either *delta encoding* or *running-length encoding* [3]. They can significantly reduce the storage size when values change slowly or remain stable over time, which is common for many spatio-temporal datasets. For non-numeric values (e.g., strings), we simply use the gzip (GNU zip) library to perform a general-purpose compression. After column-level compression, all data within a section are compressed again using gzip to

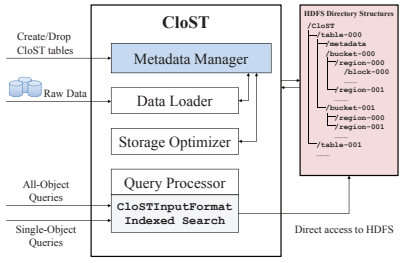


Figure 1: CloST overview

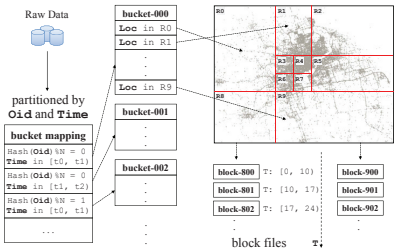


Figure 2: Hierachical partitioning

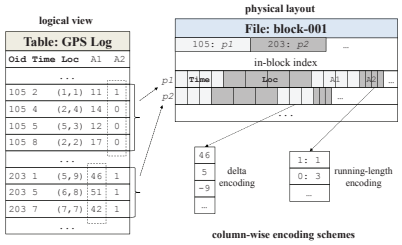


Figure 3: Block file layout

further reduce the data size. Experiment results show that our two-level compression scheme can reduce the storage size by an order of magnitude.

3.3 Metadata Management

Metadata of a table consists of definitions of common attributes, compression schemes, mappings from buckets to regions and regions to blocks, etc. CloST stores the table metadata together with the data under the same directory that is named to the table's name on HDFS. This means that the metadata storage are automatically replicated and distributed. There are two major benefits in using HDFS over using a centralized metadata storage. First, query processor directly reads metadata from HDFS and hence it does not rely on a running instance of metadata manager. Second, metadata access is unlikely to become a bottleneck of concurrent query processing. It is common that many single-object queries along with several all-object queries are issued simultaneously. Fortunately, the intensive access to metadata is dispersed to the HDFS datanodes where the metadata files are actually stored. In addition, we can easily increase the replication level of the metadata files when the system is overloaded by concurrent read of metadata.

3.4 Query Processing

For a single-object query $Q(I, S, T)$, CloST uses table metadata and in-block indexes to locate records. We first find out the buckets with regard to I and T according to level-1 partition parameters. We then locate the regions that intersects S in the above buckets according to level-2 partition indexes. In these regions, we further find out the blocks whose time range intersects T according to level-3 partition indexes. For each block file involved, we read the in-block index, get the section position corresponding to I, seek to the position and reads the whole section into memory. We assemble all records after decompressing the section data and perform the final filtering using both S and T. For an object-query $Q(S, T)$, CloST exploits MapReduce to execute the query in parallel. Initially, we finds out all the block files whose spatio-temporal range intersects S and T. Next, we starts a map task for each block file involved. Each map task sequentially scans the corresponding block file, decompresses the data and assembles the records. Then it filters each record by S and T and finally output the result.

3.5 Data Loading

The CloST data loader is designed for the typical scenario in analytical applications: a large amount of data are already present and more data are added periodically.

3.5.1 Initial Data Loading

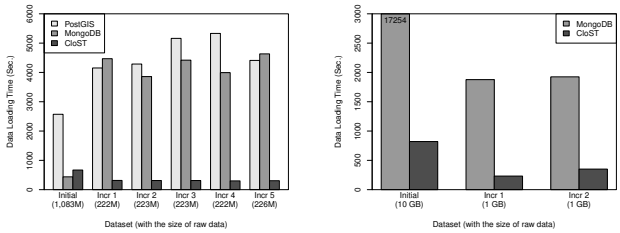
Initial data loading is performed by $n + 1$ MapReduce jobs, where n is the number of buckets. The first job partitions records into buckets and generate an index for each bucket. Each of the following jobs partitions the records of a bucket into regions then blocks and finally creates block files. Specifically, the map function of the first job emits pair ⟨`bucket_id, record`⟩ for each record extracted from the raw data, where `bucket_id` is calculated according to the level-1 partitioning policy specified in the table schema. At the reduce phase, records are grouped by `bucket_id` and the reduce function writes all the records of a group into a separate output file. Meanwhile, it maintains a sample of a group in memory to heuristically create indexes for partitioning the bucket. The indexes create roughly even partitions at both spatial and temporal dimensions, as there is no query log at this point to indicate an optimized partitioning. For each bucket, we execute a MapReduce job to create the final block files. The map function emits pair ⟨`block_id, record`⟩ for each record in the bucket. After grouping by key, the reduce function receives all records belong to the same block and then write them to HDFS using the CloST block file format. The data loading process is finished by informing the metadata manager to move the block files and the index files into the table directory.

3.5.2 Incremental Data Loading

After initial data loading, CloST performs incremental data loading when new data arrive. We first execute a MapReduce job to partition the new records into blocks using the existing indexes. These blocks are called *incremental blocks*. Another MapReduce job then merges the incremental blocks with the current blocks. Particularly, an incremental block file is merged with an existing one if they share the same spatio-temporal range. Finally, the metadata manager replaces the current block files with the merged block files and directly adds those incremental block files that are not merged with existing ones. In particular, a common assumption related to big spatio-temporal data is that data arrive in time order. With this assumption, an incremental block cannot share the same spatio-temporal range with any existing block. In such situations, our incremental data loading mechanism is very efficient because the cost of merging blocks is completely eliminated. Experiment results show that the incremental data loading of CloST is up to 20 times faster than comparison systems.

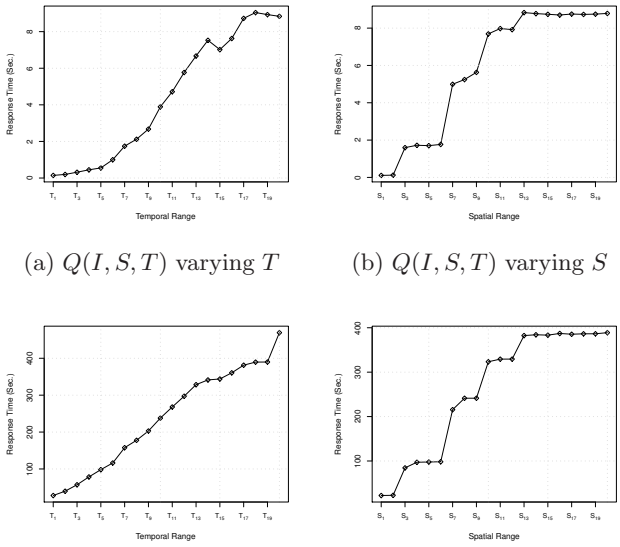
3.6 Storage Optimization

To optimize the size of block files, CloST saves the recent queries to each table and periodically tunes the spatial



(a) 1 node (b) 12 nodes

Figure 4: Data loading performance



(a) $Q(I, S, T)$ varying T (b) $Q(I, S, T)$ varying S

(c) $Q(S, T)$ varying T (d) $Q(S, T)$ varying S

Figure 5: Query Performance

and temporal partitioning based on the query log. For each block file b, the storage optimizer calculates the utilization ratio $\eta(b)$ which is defined as the average portion of records in block b contributing to the result of a query. If $\eta(b)$ is smaller than a threshold and the size of the block file is larger than twice of the minimum size, then the storage optimizer will split the block file by splitting its temporal ranges or spatially re-partition the region containing the block file. Another problem is that incremental data loading may result in a large number of small block files, if only a small amount of data is loaded. To address this issue, the storage optimizer is automatically executed after each incremental data loading to merge small block files.

4. PERFORMANCE EVALUATION

To perform evaluation, we use a cluster of 13 machines. Each machine has a single quadcore Intel Core i7-950 3.0GHz processor, 8GB DRAM, and two 2TB 7.2K RPM harddisks. Each machine runs Hadoop version 0.20.2 on Ubuntu Linux 10.04 LTS (64-bit). One of the machines is configured as both jobtracker and namenode and the others are configured as computing nodes (tasktracker and datanode). Both map and reduce slots of each computing node are set to 4.

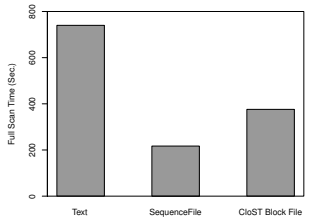


Figure 6: Full scan performance with different data format

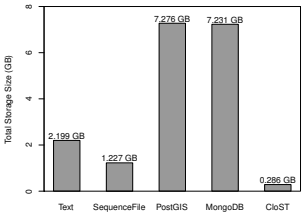


Figure 7: Storage size (including data and indexes)

The block size of HDFS is set to 256MB and each block is replicated 3 times for fault-tolerance. The dataset used in our experiments is the aformentioned GPS log dataset. The number of records is 1.9 billion and each record is about 100 bytes in text representation. The total size of the raw data files (uncompressed csv format) is 194GB.

We compare CloST with two production systems that are widely used to store spatio-temporal data. The first system is PostgreSQL 9.1 with spatial extension PostGIS 1.5.4. The other one is MongoDB 2.1.1, a high-scalable NoSQL database with built-in support of spatial indexes. In our experiments, both of them failed to load the full set of the GPS log within a reasonable time. Therefore, experiments using the full dataset are conducted with CloSTonly and we use a small portion of the dataset to evaluate our system against the above systems.

4.1 Performance of Data Loading

To evaluate the data loading performance on a single node, we use 6 datasets sampled from GPS log. The first dataset contains 10 million records and is used for initial data loading. The other 5 datasets are used for incremental data loading, each containing 2 million records. For PostGIS and MongoDB, the initial data loading loads data into an empty table creates the necessary indexes. The subsequent data loading are performed in the presence of these indexes. As is shown in Figure 4(a), the CloST data loading speed is up to 20 times faster than the other two systems. As a heavy-weight RDBMS, PostGIS is slow at both data loading phases. MongoDB is the fastest system performing initial data loading. However, the presence of indexes dramatically affects its data loading speed. We use a 10GB initial dataset with two 2GB incremental dataset to compare the parallel data loading performance and describe the result in Figure 4(b). The results show that the CloST loads data at least 5 times faster than MongoDB.

4.2 Performance of Query Processing

We evaluate the query performance of CloST using the full dataset. To reveal the relation between query response time and the range size, we select a number of spatial ranges and temporal ranges. Specifically, S_1 through $S20$ are 20 spatial bounding boxes having the same centroid and similar shape and ordered by the increasing order of their areas (sizes); T_1 through T_{20} are 20 time ranges with the same start time and ordered by their end time.

Figure 5 shows the query response time in CloST. Both single-object query and all-object exhibits the same trend when the query range changes. In Figure 5(a) and Fig-

ure 5(c), the spatial range is unbounded and the time range varies from T_1 to T_{20}. The response time yields a linear increasing trend as the size of the time range increases. In Figure 5(b) and Figure 5(d), the time range is unbounded and only the spatial range varies. The response time exhibits a different pattern in that there are several jump points when the spatial range grows. The reason is that the GPS records are heavily skewed on the spatial dimension; whereas they are relatively more evenly distributed on the temporal dimension. Figure 6 shows the performance of full scan using different file format. Full scan over CloST block files is 2 times faster than scanning the raw data files, while it is around 1/3 slower than using the Hadoop `SequenceFile` format. However, `SequenceFile` does not support range scan and consumes much more storage space.

4.3 Performance of Data Compression

Figure 7 shows the storage size of using various approaches to store the GPS log dataset. For the 2GB dataset, CloST uses only 0.286GB storage space, which is 13.0% of the raw data, 22.3% of the `SequenceFile` size, 3.9% of the space used by PostGIS, and 4.0% of the space used by MongoDB. For the full dataset, the storage size in CloST is 21.22GB and the compression ratio is 10.9% which is even higher than the compression ratio of using gzip to compress the raw data files (14.4%). The high compression ratio is an important advantage of using CloST to store big spatio-temporal data.

5. RELATED WORK

Google designed and implemented GFS [5] to enable efficient sequential scan of large unstructured datasets. GFS effectively handles node and network failures and is able to manage thousands of machines. HDFS [7] is an open-source analogue GFS and is adopted by Facebook to store dozens of petabytes of data. They are the most widely used underlying storage systems for MapReduce [2]. Beyond file-level storage, some big data storage systems such as mongoDB [8] supports secondary spatial indexes.

There has been many data structures proposed for indexing spatial and spatio-temporal data including Quadtree [9], kd-tree [10], R-tree [11], TB-tree [12], SEB-tree [13], etc. These literatures have little discussion about how data are actually stored on disk. To address the storage issues with relatively large (several gigabytes) spatio-temporal datasets, TrajStore [3] and PIST [4] attempt to cluster data according to spatial and temporal proximities. Both systems only consider non-distributed environments and therefore cannot trivially scale to datasets of several terabytes.

6. CONCLUSION

In this paper, we presented CloST, a Hadoop-based storage system for big spatio-temporal data analytics. It is targeted at fast data loading, scalable spatio-temporal range query processing and efficient storage usage to handle very large datasets. CloST uses a novel data model which takes into account three core attributes including an object id, a spatial location and a time to physically organize the data. Data are partitioned hierarchically into blocks using all the core attributes. Each block is stored as a regular file on HDFS with a very compact format. We also proposed scalable data loading and query processing algorithms. The experiment results demonstrated that CloST is suitable for big spatio-temporal data analytics.

7. ACKNOWLEDGEMENT

This research was supported in part by Hong Kong, Macao and Taiwan Science & Technology Cooperation Program of China under Grant No. 2012DFH10010.

8. REFERENCES

[1] Apache Hadoop. `http://hadoop.apache.org`.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[3] Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 109–120. IEEE, 2010.

[4] Viorica Botea, Daniel Mallett, Mario A. Nascimento, and Jörg Sander. Pist: An efficient and practical indexing technique for historical spatio-temporal point data. *Geoinformatica*, 12(2):143–168, June 2008.

[5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[6] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1199–1208, Washington, DC, USA, 2011. IEEE Computer Society.

[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 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.

[8] mongoDB. `http://www.mongodb.org`.

[9] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, June 1984.

[10] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[11] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[12] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches to the indexing of moving object trajectories. In *Proc. 26th VLDB conf.*, pages 395–406, 2000.

[13] Zhexuan Song and Nick Roussopoulos. Seb-tree: An approach to index continuously moving objects. In *Proceedings of the 4th International Conference on Mobile Data Management*, MDM '03, pages 340–344, London, UK, UK, 2003. Springer-Verlag.