An Efficient Distributed Index for Geospatial Databases

Le Hong Van[1](✉) and Atsuhiro Takasu[1,2]

[1] SOKENDAI (The Graduate University for Advanced Studies), Kanagawa, Japan
{l-van,takasu}@nii.ac.jp
[2] National Institute of Informatics, Tokyo, Japan

Abstract. The recent and rapid growth of GPS-enabled devices has resulted in an explosion of spatial data. There are three main challenges for managing and querying such data: the massive volume of data, the need for a high insertion throughput and enabling real-time spatial queries. Although key–value store databases handle large-scale data effectively, they are not equipped with effective functions for supporting spatial data. To solve this problem, we propose an efficient spatial index structure based on HBase, a standard key–value store database. We first use Geohash as the rowkey in HBase to sustain high insert throughput. We present a novel data structure, the binary Geohash rectangle-partition tree, that partitions data into subrectangles, then add these subrectangles into an R-Tree to support spatial queries. Our experiments demonstrate high scalability and an improved performance with spatial queries, when compared with a state-of-the-art system. They also show a good real-time query-processing capability, with response times of less than one second.

Keywords: HBase · GeoHash · R-Tree · Spatial index · Real-time

1 Introduction

With the rapid growth of spatial data, support for high-performance spatial queries about large-scale data is becoming essential in both scientific research and daily life. For example, intelligent transportation systems such as advanced traffic management systems are exploiting the massive volume of real-time sensor data from probe cars and GPS-enabled devices to provide efficient traffic balancing and trajectory recommendation services. Commuters can take advantage of these systems by sending data continuously about their current location and then receiving relevant analyses of their real-time traffic situation.

The main challenges for these systems are the massive volume of data, the need for a high insertion throughput and support for real-time spatial queries. For instance, traffic management systems in busy cities such as Tokyo have to support a large number of location updates per minute from probe cars such as

© Springer International Publishing Switzerland 2015
Q. Chen et al. (Eds.): DEXA 2015, Part I, LNCS 9261, pp. 28–42, 2015.
DOI: 10.1007/978-3-319-22849-5_3

taxis and buses and from collaborating private cars and GPS-enabled devices. At the same time, they have to analyze the data and respond in real time.

To meet these requirements, systems need database management systems (DBMS) that have good scalability while guaranteeing satisfactory performance with real-time spatial queries. Although distributed-computing technologies such as Hadoop MapReduce [8] are reasonable choices for handling large volumes of data, with some frameworks based on MapReduce such as SpatialHadoop [9] supporting spatial queries, they have high latency in comparison with a real-time system. Key–value store databases such as HBase, 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 queries.

Some methods [13,16] use Geohash [2], a linearization technique that transforms two-dimensional spatial data into a one-dimensional hashcode, to handle spatial data in HBase. However, they do not address edge cases and the Z-order limitations of Geohash, which can lead to incorrect results and can decrease the performance on spatial queries. In this paper, we tackle the limitations of Geohash with spatial data by proposing an efficient distributed spatial index (Fig. 1), which uses Geohash and the R-Tree [12] with HBase. We use Geohash as the key for key–value pairs for HBase, then further utilize the R-Tree, a multi-dimensional index structure for geographical data, to generate correct results and improve query performance. To bridge Geohash and the R-Tree, we propose a novel data structure, the binary Geohash rectangle-partition tree (BGRP Tree). By using the BGRP Tree, we partition a Geohash range into nonoverlapping subrectangles, then insert these subrectangles into the R-Tree.

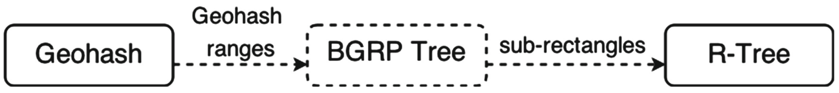


Fig. 1. Two-tier index of Geohash and the R-Tree using the BGRP Tree to bridge the tiers

Using Geohash as the key for HBase enables geospatial databases to achieve a high insert throughput because Geohash encoding is not computation intensive. The R-Tree index helps improve the performance and the accuracy of spatial queries. Our experimental results indicate that querying that can take advantage of the Geohash and R-Tree index outperformed both querying that used only Geohash on HBase and querying in a MapReduce-based system. We also observed that our proposed index could process queries in real time, with response times of less than one second.

This paper is organized as follows. Section 2 briefly surveys related work. The design of our proposed index and the new data structure, the BGRP Tree, are presented in Sects. 3 and 4, respectively. Experimental evaluation is discussed in Sect. 5, followed by our conclusions and plans for future work.

2 Related Work

Managing and querying spatial data can be processed efficiently using traditional relational DBMS such as Oracle Spatial and PostGIS [4]. However, these systems cannot meet the major requirements of a geospatial system, namely, scaling and analyzing a large volume of data, because of insufficient scalability.

When considering scalable data-processing systems, systems based on MapReduce [8] such as Hadoop [19] have dominated. Hadoop provides not only scalability to petabytes of data but is also fault tolerant and highly available. However, Hadoop is not aimed primarily at supporting the processing and analysis of spatial data. There are some extensions, such as SpatialHadoop [9] and Hadoop-GIS [5], which do support high-performance spatial queries based on the MapReduce framework.

SpatialHadoop is an extension of Hadoop that is designed with a view to handling massive volumes of spatial data. It comprises four components: a high-level spatial language, spatial operations, MapReduce and storage. The key idea in SpatialHadoop is a two-tier index (global and local) that uses an R-Tree to index spatial data inside the Hadoop distributed file system (HDFS). When compared with the original Hadoop, R-Tree indexing in SpatialHadoop performs better with spatial data. However, because the R-Tree is a balanced search tree, rebalancing is necessary after a number of data insertions. With continuous insertion from spatial data sources such as sensor data, this rebalancing can be computationally expensive.

Hadoop-GIS is another scalable and high-performance spatial data processing system, which is integrated into Hive [18] to support declarative spatial queries. Like SpatialHadoop, Hadoop-GIS has two R-Tree index levels. The local index in Hadoop-GIS is built on the fly and is therefore flexible. However, because the MapReduce framework is suited to a batch-processing context, Hadoop-GIS has a high latency in the context of a real-time system.

Spatial support has been also extended to NoSQL-based solutions. For instance, MD-HBase [15] is a scalable multidimensional database for location-aware services, which layers a multidimensional index over the key–value store. It uses linearization techniques such as Z-ordering [14] to transform multidimensional data into one-dimensional form, then stores the data in HBase. It also layers multidimensional index structures, such as K-d trees [6] and quad-trees [10], on top of the key–value stores. For our work, we adopt a different approach by using Geohash as a linearization technique and the R-tree as a multidimensional index. The common prefix in Geohash offers a better way to find nearest neighbors and the R-Tree can accelerate a nearest-neighbor search.

Lee et al. [13] and Pal et al. [16] also adopt Geohash to handle spatial data in HBase. In [13], spatial queries with the Geohash index performed better than spatial queries with the R-Tree index. However, they did not present solutions for the limitations of Geohash.

3 Distributed Spatial Index

3.1 Basis of Distributed Spatial Index

Apache HBase [11] is a distributed scalable database that takes advantage of the HDFS. Because HBase stores data as a sorted list of key–value pairs, it is also a key–value store database. Within a table, data are stored 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.

One of the advantages of HBase is its auto-sharding capability, which means it can dynamically split tables when the quantity of data becomes excessive. The basic unit of scalability and load balancing in HBase is called a region. Because HBase keeps rowkeys in lexicographical order, each region stores a range of rowkeys between a start rowkey and an end rowkey.

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, which was invented by Gustavo Niemeyer in 2008 [2], provides a solution to transform longitude/latitude coordinates into unique geocodes.

Geohash has some notable properties. It is a hierarchical spatial data structure that subdivides space into grid-shaped buckets. Points that share the same prefix will be close to each other geographically, and all Geohashes with the same prefix are included in the rectangle for that prefix. The shorter a Geohash is, the larger will be its rectangle. Geohash has a Z-order traversal of rectangles in each resolution (Fig. 2). To best take advantage of Geohash, we store the spatial data in HBase where the rowkeys are Geohashes. In this way, points that are close geographically are stored close together.

Despite its advantages, Geohash has some limitations.

Edge-Case Problem. Locations on the opposite sides of the 180-degree meridian are close to each other geographically but have no common prefix in Geohash codes. Points close together at the North and South poles may also have very different Geohashes.

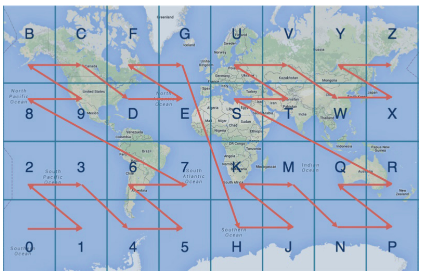


Fig. 2. Z-order traversal in Geohash

Z-Order Problem. This problem is related to Z-order traversal. Some points having a common Geohash prefix are not always close to each other geographically. For instance, in Fig. 2, locations in rectangles whose Geohash codes are *7* and *8* will be stored close together, but are not close geographically. In contrast, nearby locations such as in rectangles *7* and *e* have widely separated Geohash codes.

3.2 Index Design

Although using Geohash as the rowkey in HBase helps the system support high insert throughput, using Geohash alone does not guarantee efficient spatial-query processing. For instance, nearby points often share the same Geohash prefix, enabling us to find the k nearest neighbors (kNN) of a query point using a prefix filter. However, the edge-case problem would lead to insufficient or incorrect results when the query point has a longitude near 180 or –180 degrees. The Z-order problem can cause redundant scanning. For example, in Fig. 2, when the system wants to scan all points ranging from rectangles with Geohash *1* to *4*, it will scan rectangles *1*, *4* and the unrelated rectangles *2* and *3*.

To obtain accurate query results and prune unnecessary scanning, we use the R-Tree as a secondary index tier. The R-tree stores only rectangles, rather than points. In this way, we can avoid frequent rebalancing of the tree, which is a time-consuming process, even if new points arrive continuously. In our method, we first partition regions into rectangles using the *longest common prefix* (LCP). Because there is overlap between these rectangles, we use the BGRP Tree (described in Sect. 4) for further partitioning of rectangles into subrectangles until there is no overlap between them. Finally, all subrectangles are inserted into the R-Tree. When processing spatial queries, we find the rectangles in the R-Tree that may contain query results before scanning. We then scan only the found rectangles, thereby pruning the scanning on unrelated regions.

As described in Sect. 3.1, the data are split into regions dynamically. HBase performs a finer-grained partition for the denser places, i.e., those that contain more points in a small area. The R-tree is constructed based on these HBase partitions and the proposed index can therefore manage any data skew by this adaptive partitioning.

All the steps for R-Tree creation are processed in the background, removing the need for R-Tree creation overhead when processing queries. Because the R-Tree is stored in memory and the number of nodes in the tree is not large, the R-Tree search overhead is also very small. We evaluate this overhead empirically in Sect. 5.

3.3 LCP-Based Region Partition

We partition regions by using the LCP of the Geohash codes. Algorithm 1 describes the algorithm for region partition. We first find the LCP of the start and end rowkeys. We then obtain the character next to the LCP from the start rowkey (c_1) and the end rowkey (c_2). Geohash uses base32 [1] to encode spatial points. We concatenate the LCP with each character from c_1 to c_2 in base32 to create the Geohash of a new rectangle. By using this algorithm, we can ensure that the list of rectangles will cover all points in the region.

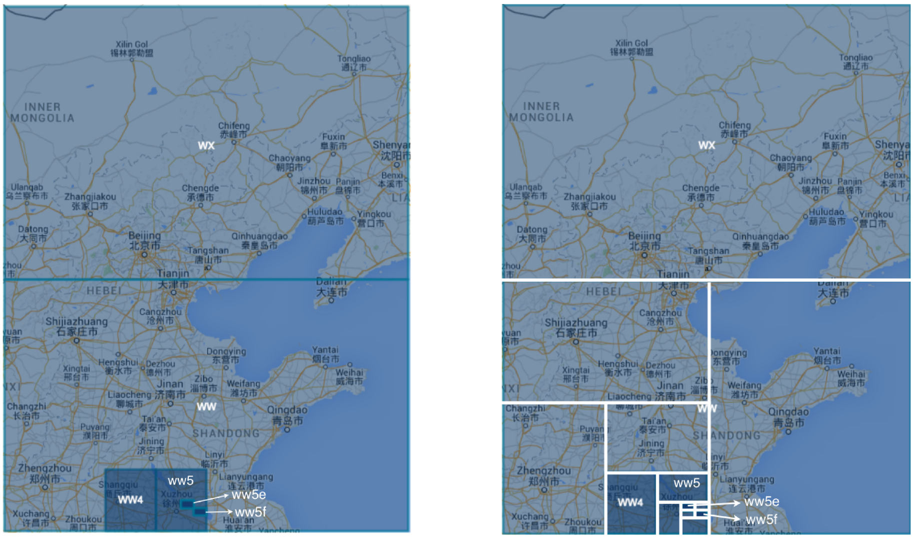
Table 1 shows an example of region partitioning. In this example, we have three regions in the Geohash range from *ww4durf3yp21* to *wx1g1zbnwxnv*. The result of region partitioning is a list of rectangles that cover all points in the three regions. Figure 3a shows the results of the region partitioning on a map. In this figure, we can assume that the areas involving rectangles with longer Geohashes, i.e., *ww5e* and *ww5f*, have higher point density than other areas.

Algorithm 1. Region Partitioning

```
[1] $lcp \longleftarrow findLongestCommonPrefix(startRowKey, endRowKey)$
[2] $lengthOfLcp \longleftarrow length(Lcp)$
[3] $c_1 \longleftarrow startRowKey[lengthOfLcp]$
[4] $c_2 \longleftarrow endRowKey[lengthOfLcp]$
[5] /* Loop on list of symbols of base32                    */
[6] for $c \leftarrow c_1$ to $c_1$ do
[7]     $addToResult(lcp + c)$
```

Table 1. An example of region partitioning

Region	Start Rowkey	End Rowkey	LCP	Rectangles
Region 1	**ww**4durf3yp21	**ww**5eh56ebb98	ww	ww4, ww5
Region 2	**ww5**eh56ebb98	**ww5**ffn87crgs	ww5	ww5e, ww5f
Region 3	ww5ffn87crgs	**w**x1g1zbnwxnv	w	ww, wx



(a) Rectangles generated from the example in Table 1

(b) Rectangles after using the BGRP Tree for partitioning

Fig. 3. Partition using the BGRP tree

4 The BGRP Tree

In Fig. 3a, some rectangles overlap, such as *ww*, *ww5* and *ww5e*. Inserting these rectangles directly into the R-Tree would lead to redundancy in the query results. For instance, when searching the R-Tree for a rectangle containing a point with prefix ww5e, the results would be ww, ww5 and ww5e.

The BGRP Tree is a data structure for representing Geohash rectangles. A BGRP Tree satisfies the following properties.

- The level of a node is the length of the Geohashes.
- All nodes (rectangles) are within the Geohash range.
- All nodes have between 1 and 32 children.
- There is no overlap between nodes.
- The leaves cover the whole surface of the original rectangles.
- Tree construction does not depend on the order of rectangle insertion.

4.1 The BGRP Task

A BGRP task is necessary in many insertion cases, as explained in Sect. 3. The input for this task is two rectangles, with the bigger rectangle containing the smaller one. Both could be presented by a Geohash or a range of Geohashes. The output of this task is a set of subrectangles that includes the smaller rectangle and covers all the surface of the bigger one. Our BGRP method recursively subdivides a rectangle into two subrectangles until one of the two subrectangles matches the smaller input rectangle. This method is similar to the binary space-partitioning method [17].

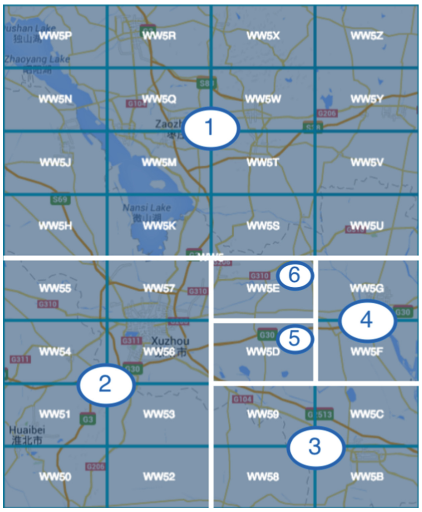


Fig. 4. The BGRP task

Figure 4 describes the BGRP task for the inputs *ww5* and *ww5e*. First, we divide *ww5* into two parts, a subrectangle sr_1 with the Geohash from *ww50* to *ww5g* and a subrectangle sr_2 with the Geohash from *ww5h* to *ww5z*. The rectangle sr_2 is inserted as is, whereas sr_1 is split further into two subrectangles. The process is recursively applied until one of the subrectangles is equal to *ww5e*. As a result, six subrectangles are obtained (Fig. 4).

Each rectangle in the BGRP Tree corresponds to a range of Geohashes. For instance, the Geohash range of sr_2 is from *ww5h* to *ww5z*, whereas *ww5* corresponds to a range from *ww50* to *ww5z*. For a rectangle whose Geohash range length is r, the task will require $log_2 r$ binary partition steps and will result in a set of $log_2 r + 1$ subrectangles.

4.2 BGRP Tree Searching

The search algorithm descends the tree from the root, recursively searching subtrees for the highest-level node that contains the input rectangle, as shown in Algorithm 2. Like search algorithms in R-Trees, if a node does not contain the query point, we do not search inside that node. If there are n nodes in the tree, the search requires $O(log(n))$ time.

Algorithm 2. BGRP Tree Search

```
search(node, rect)
    /* this function calls initially with node = root          */
    input  : node, rect
    output: highest-level node that contains input rectangle
[1]     if $node = rect$ then
[2]         return node
[3]     if $node$ is leaf then
[4]         return node
[5]     else
[6]         for $child \in children$ of $node$ do
[7]             if $child = rect$ then
[8]                 return child
[9]             else if child contains rect then
[10]                return search(child, rect)
```

4.3 BGRP Tree Insertion

Insertion is the most complicated aspect of the BGRP Tree. Figure 5 shows the insertion of all rectangles in Fig. 3a into a BGRP Tree. The resulting tree is shown in Fig. 3b.

Algorithm 3 describes the insertion algorithm for the BGRP Tree. If the tree has been initialized, the first step is to search for the highest-level node containing the insert rectangle (as described in Sect. 4.2). If no node is found, we create a new root by obtaining the LCP between the Geohash of the root and the rectangle. For instance, in Fig. 5, when inserting *ww5*, the LCP between

Algorithm 3. BGRP Tree Insertion

```
[1]  if no root then
[2]      $root \longleftarrow rect$
[3]      return
[4]  $containNode \longleftarrow search(root, rect)$
[5]  if containNode is null then
[6]      $oldRoot \longleftarrow root$
[7]      $root \longleftarrow getLongestCommonPrefix(oldRoot, rect)$
[8]      $insertWithoutPartition(root, oldRoot)$
[9]      $insertWithoutPartition(root, rect)$
[10] else
[11]     if $containNode = rect$ then
[12]         if containNode is not leaf and is not yet fully partitioned then
[13]             $binaryNodePartition(containNode)$
[14]     else
[15]         if containNode is leaf then
[16]             $binaryLeafPartition(containNode, rect)$
[17]         else
[18]             $insertWithoutPartition(containNode, rect)$
```

the Geohash of the current root (*ww4*) and *ww5* is *ww*. We therefore create the new root *ww*, then insert *ww5* into the new root without partitioning.

There are two types of insertion in the BGRP Tree, namely insertion with a binary partition and insertion without partition. Insertion without partition is applied when inserting into an existing nonleaf node. We create a binary partition when inserting into a leaf, as shown for the insertion of *ww5e* and *ww5f* in Fig. 5. Binary partitioning is also applied when the insert rectangle exists in the tree as a nonleaf node that is not yet fully partitioned. Insertion of *ww* in Fig. 5 is an example of this case. Before being inserted, *ww* was a nonleaf node with only two children, *ww4* and *ww5*, which do not cover the entire surface of *ww*. Therefore, we need to partition *ww* to ensure that the surface of *ww* is completely covered in leaves.

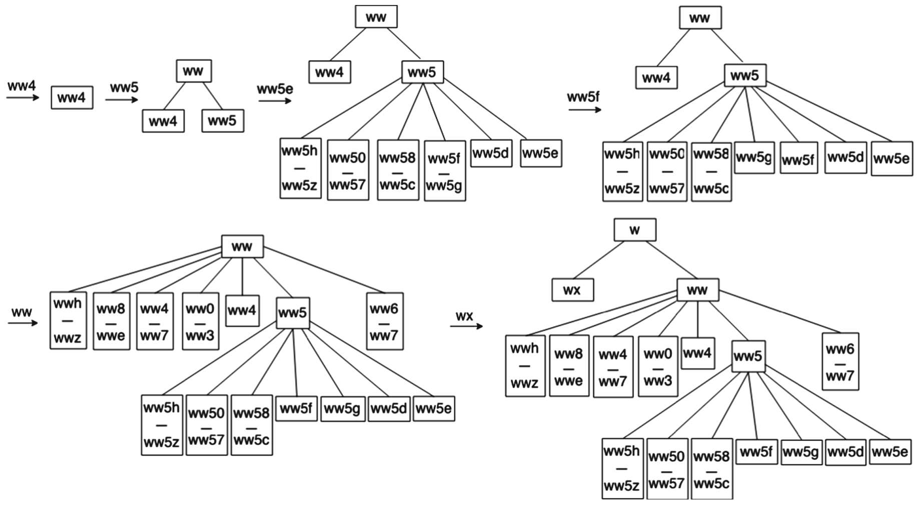


Fig. 5. BGRP tree insertion

Binary partitioning is different for leaf and nonleaf nodes. For a nonleaf node, the existing children of the node need to be considered and we therefore partition the nonleaf node until all its children are contained in the results.

5 Experimental Evaluation

5.1 Experimental Setup

HBase. We built a cluster comprising one HMaster, 60 Region Servers and three Zookeeper Quorums. 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 used Apache HBase 0.98.7 with Apache Hadoop 2.4.1 and Zookeeper 3.4.6. Replication was set to two.

SpatialHadoop. To evaluate the performance of our method, we compared it with SpatialHadoop. For this, we used a cluster with one master and 64 slaves. Each node had one virtual core, 8GB memory and a 64GB hard drive with CentOS 7.0 (64-bit). Replication was also set to two. We installed SpatialHadoop v2.1, which shipped with Apache Hadoop 1.2.1.

5.2 Dataset

We used two real-world datasets, namely T-Drive [20,21] and OpenStreetMap (OSM) [3].

T-Drive. T-Drive was generated by 30,000 taxis in Beijing over a period of three months. The total number of records in this dataset is 17,762,390. It requires about 700 MB, and was split into 16 regions for the HBase cluster.

OSM. OSM is an open dataset that was built by contributions from a large number of community users. It contains spatial objects presented in many forms such as points, lines and polygons. We used the dataset of nodes for the whole world. It is a 64 GB dataset that includes 1,722,186,877 records. By using Snappy compression, we inserted the OSM dataset into 251 regions on the HBase cluster.

5.3 Comparison Method

To evaluate the performance of spatial queries based on our proposed index, we executed kNN queries using the two datasets, T-Drive and OSM. We evaluated three kNN query-processing methods using HBase and compared them with SpatialHadoop as a baseline method. The three methods are described below.

HBase with Geohash. We used Geohash as the rowkey for storing data in HBase. When processing a kNN query, the client first sends a scan request with a prefix filter to the region servers, obtains the scan results, then calculates kNN results for the received data. The prefix filter enables scanning of only those points that share the prefix with the query point.

HBase Parallel with Geohash. HBase supports efficient computational parallelism by using "coprocessors". The idea of HBase coprocessors was inspired by Google's BigTable [7] coprocessors. By using coprocessors, we could apply a divide-and-conquer approach for the kNN query. In parallel kNN, the kNN is processed inside each of the regions. In this way, instead of returning the unprocessed scan results, each region returns the kNN for its region to the client.

HBase Parallel with Geohash and R-Tree. With our proposed index design, the client searches the R-Tree for the rectangle that contains the query point before scanning. If the number of neighbors inside that rectangle is insufficient, the client continues the scan inside the neighbors of the rectangle until sufficient results are found.

SpatialHadoop. We also experimented with kNN queries using SpatialHadoop for comparison with our method.

We chose two groups of points for the experiments, namely, a group of high-density points and a group of low-density points. High-density points are the points in crowded areas, which will contain many points. Conversely, low-density points are the points in uncongested areas, which contain few points. Because HBase can handle data skew via its auto-sharding capability, we visualized the start rowkeys and end rowkeys for all regions on the map, then randomly chose points in high-density and low-density areas. In low-density areas, when k is large, a query may not find enough neighbors in its early stages. In such cases, we continue by scanning an increasingly large area (by decreasing the length of the prefix in the prefix filter) until sufficient results are retrieved. The query-processing time is measured by calculating the elapsed time between when the system starts scanning data and when all results are returned.

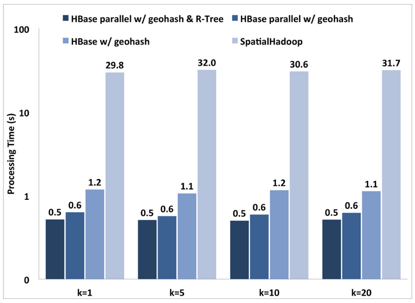
5.4 Results and Discussion

Insert Efficiency. We evaluated the insert performance for both the T-Drive and OSM datasets. Table 2 shows the insertion-process efficiency in terms of the number of inserted points per second. It is calculated by dividing the number of inserts by the elapsed time. This experiment involved insertion from a single client. Evaluation of insertion from multiple clients is proposed for future work. As shown in the table, our index structure can process about 80,000 inserts per second. Note that, unlike other systems with a tree-structured index, no extra tree balancing is needed even for heavy inserts. Because creating rectangles to insert into the R-Tree is based on the start and end rowkeys of regions, recreating the R-Tree is only necessary when a region exceeds a predefined maximum size and is partitioned into two regions.

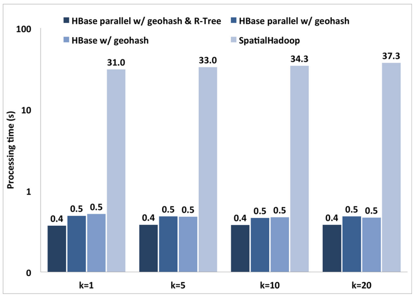
kNN Queries for the T-Drive Dataset. Figure 6 shows the performance of kNN queries for the T-Drive dataset. We see that parallel kNN using Geohash and R-Tree outperforms all other kNN query methods. In this experiment, note that our index design operates about 60 to 90 times faster than SpatialHadoop. 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 has to send the procedure to slave servers for every query, thereby taking more time for network communication.

Table 2. Insert throughput

Dataset	Number of inserts	Elapsed time (sec)	Throughput (inserts/sec)
T-Drive	17,762,390	399	44,517
OSM	1,722,186,877	21,268	80,975

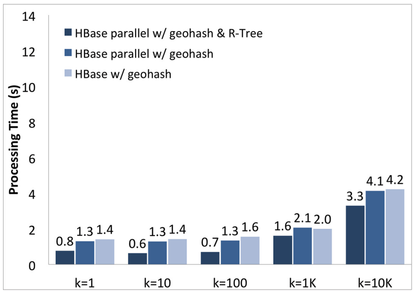


(a) High-density points

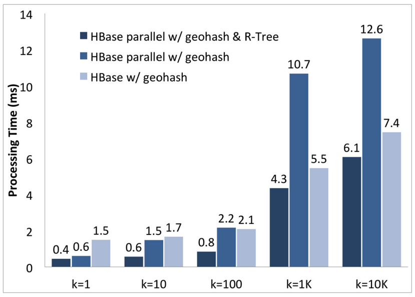


(b) Low-density points

Fig. 6. Performance of kNN queries for the T-Drive dataset



(a) High-density points

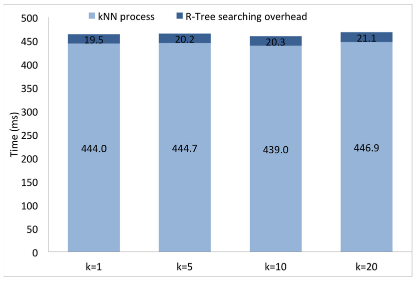


(b) Low-density points

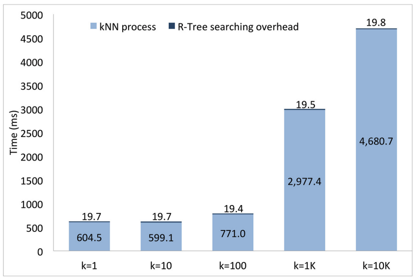
Fig. 7. Performance of kNN queries for the OSM dataset

kNN Query for the OSM Dataset. As Fig. 7 shows, when processing the larger dataset, a kNN query with the proposed two-tier index also outperforms other query methods. For parallel kNN query with Geohash, I/O bottlenecks may occur because regions that do not contain neighbors of the query point still send responses to the client (with no useful results). The T-Drive data spread across only 16 regions, giving a less-than-critical I/O load. However, for the larger OSM dataset, with its number of regions increased 15-fold, the client I/O load is much heavier. For our proposed method, where we search the R-Tree before scanning to limit the regions requiring processing, we can reduce this I/O load.

Unlike the experiments using the T-Drive dataset, experiments with the OSM dataset differed in performance, depending on whether high-density or low-density points were being processed. For low-density points and larger values of k, the queries could not find sufficient neighbors during the initial search. Therefore, queries had to search again over a larger area, which led to higher latency. By using the R-Tree to search rectangles near the query point, the queries could reduce the scanning of unrelated areas, thereby achieving an improved performance.



(a) T-Drive dataset



(b) OSM dataset

Fig. 8. R-Tree searching overhead (Color figure online)

R-Tree Searching Overhead. We also measured the overhead involved in the R-Tree searching step. Figure 8 shows the performance of parallel kNN queries using the proposed index with an R-Tree searching overhead. The bars in darker color represent the cost of searching the R-Tree before scanning, and the bars in lighter color represent the kNN query processing time (as explained in Sect. 5.3). As shown in the figure, the cost for the R-Tree searching is very small (around 20 milliseconds). Instead of inserting points from the whole dataset, we insert only big rectangles representing dataset partitions into the R-Tree, implying only a small number of nodes and a correspondingly small cost for the R-Tree searching. Furthermore, because the R-Tree is small, we can store it in memory and reduce the reading-time latency.

6 Conclusion

In this paper, we proposed a two-tier index for spatial data in a key–value store database using Geohash and an R-tree structure. We also defined the BGRP Tree data structure to bridge the two tiers. Using our proposed index, HBase can support spatial queries efficiently. Experimental results for two real-world datasets demonstrated the performance improvement for spatial queries when using the two-tier index with Geohash and an R-Tree. The results also showed that the R-Tree searching overhead is very small and the response time for queries would meet the requirements of real-time systems.

In future work, we plan to consider other complex spatial queries such as range queries, spatial joins and convex hulls. For some applications such as route planners based on current traffic data, a spatiotemporal database plays an important role in the data analysis. Spatiotemporal indexing for a key–value store database is one of our ongoing projects. We are also working on stream data, such as data being collected continuously from sensors. Stream data pose many new challenges, such as processing data when its spatial relationships are changing continuously and maintaining consistency while guaranteeing high read/write throughput and effective query processing.

Acknowledgments. This work was partly supported by the research promotion program for national-level challenges Research and development for the realization of next-generation IT platforms? by MEXT, Japan and the Strategic Innovation Promotion Program of the Japanese Cabinet Office.

References

1. Base32. http://en.wikipedia.org/wiki/Base32
2. GeoHash. http://geohash.org
3. Openstreetmap. http://www.openstreetmap.org
4. PostGIS. http://postgis.net
5. Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., Saltz, J.: Hadoop GIS: a high performance spatial data warehousing system over mapreduce. Proc. VLDB Endow. **6**(11), 1009–1020 (2013)
6. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
7. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) **26**(2), 4 (2008)
8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
9. Eldawy, A., Mokbel, M.F.: A demonstration of spatialhadoop: an efficient mapreduce framework for spatial data. Proc. VLDB Endow. **6**(12), 1230–1233 (2013)
10. Finkel, R.A., Bentley, J.L.: Quad trees a data structure for retrieval on composite keys. Acta informatica **4**(1), 1–9 (1974)
11. George, L.: HBase: The Definitive Guide. O'Reilly Media Inc., Sebastopol (2011)
12. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching, vol. 14. ACM, New York (1984)
13. Kisung, L., Ganti, R.K., Srivatsa, M., Liu, L.: Efficient spatial query processing for big data. Framework **7**(11), 4–12 (2014)
14. Morton, G.M.: A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. International Business Machines Company, New York (1966)
15. Nishimura, S., Das, S., Agrawal, D., Abbadi, A.E.: MD-HBase: a scalable multi-dimensional data infrastructure for location aware services. In: 2011 12th IEEE International Conference on Mobile Data Management (MDM), vol. 1, pp. 7–16. IEEE (2011)
16. Pal, S., Das, I., Majumder, S., Gupta, A.K., Bhattacharya, I.: Embedding an extra layer of data compression scheme for efficient management of big-data. In: Mandal, J.K., Satapathy, S.C., Sanyal, M.K., Sarkar, P.P., Mukhopadhyay, A. (eds.) Information Systems Design and Intelligent Applications, pp. 699–708. Springer, India (2015)
17. Schumacker, R.A., Brand, B., Gilliland, M.G., Sharp, W.H.: Study for applying computer-generated images to visual simulation. Technical report, DTIC Document (1969)
18. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a map-reduce framework. Proc. VLDB Endow. **2**(2), 1626–1629 (2009)
19. White, T.: Hadoop: The Definitive Guide. O'Reilly Media Inc., Sebastopol (2012)

20. Yuan, J., Zheng, Y., Xie, X., Sun, G.: Driving with knowledge from the physical world. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 316–324. ACM (2011)
21. Yuan, J., Zheng, Y., Zhang, C., Xie, W., Xie, X., Sun, G., Huang, Y.: T-drive: driving directions based on taxi trajectories. In: Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 99–108. ACM (2010)