

Quadrant-Based MBR-Tree Indexing Technique for Range Query Over HBase

Bumjoon Jo and Sungwon Jung^(✉)

Department of Computer Science and Engineering,
Sogang University, 35 Baekbeom-ro, Mapo-gu, Seoul 04107, Korea
{bumjoonjo, jungsung}@sogang.ac.kr

Abstract. HBase is one of the most popular NoSQL database systems. Because it operates on a distributed file system and supports a flexible schema, it is suitable for dealing with large volumes of semi-structured data. However, HBase only provides an index built on one dimensional rowkeys of data, which is unsuitable for the effective processing of multidimensional spatial data. In this paper, we propose a hierarchical index structure called a Q-MBR (quadrant based minimum bounding rectangle) tree for effective spatial query processing in HBase. We construct a Q-MBR tree by grouping spatial objects hierarchically through Q-MBRs. We also propose a range query processing algorithm based on the Q-MBR tree. Our proposed range query processing algorithm reduces the number of false positives significantly. An experimental analysis shows that our method performs considerably better than the existing methods.

Keywords: HBase · NoSQL · Spatial data indexing · Q-MBR tree · Range query

1 Introduction

In recent years, a number of studies have attempted to use Hadoop distributed file system and MapReduce framework to deal with spatial queries on big spatial data [1–3]. However, these methods suffer from a large amount of data I/O during query processing because of a lack of spatial awareness of the underlying system. In order to overcome this problem, there have been attempts to distribute spatial data objects over cloud infrastructure by considering their spatial proximity [4–7]. SpatialHadoop [4] and Hadoop-GIS [5] observe the spatial proximity of data, and store adjacent data into same storage block of Hadoop. They provide global index to retrieve relevant blocks for query processing and also provide local index to explore data in each blocks. Dragon [6] and PR-Chord [7] use similar indexing techniques on P2P environment. The limitation of these methods is that they are vulnerable to update. When the updating is issued, distribution of data is changed and entire index structure should be modified.

As an alternative to methods based on the Hadoop system, there have been several studies have enhanced the spatial awareness of NoSQL DBMS, especially HBase [8–10]. HBase provides an effective framework for fast random access and updating of data on a distributed file system. Because HBase only provides an index built on one dimensional rowkeys of data, most studies attempt to provide a secondary index of

spatial data in the HBase table format. However, because these are not designed to fully utilize the properties of HBase, inefficient I/O occurs during spatial query processing.

In this paper, we propose indexing and range query processing techniques to efficiently process large spatial data in HBase. Our proposed indexing method adaptively divides the space into quadrants like a quad tree, by reflecting the data distribution, and creates an MBR in each quadrant. These MBRs are used to construct a secondary index to access spatial objects. The index is stored as an HBase table, and accessed in a hierarchical manner.

This paper is organized as follows. Section 2 describes our data partitioning method, named Q-MBR, and the index structure that employs it. Section 3 describes the algorithms for insertion and range query using the Q-MBR tree. In Sect. 4, we experimentally evaluate the performance of our index and algorithms. Finally, Sect. 5 concludes the paper.

2 Spatial Data Indexing Using Quadrant-Based MBR

2.1 Data Partitioning with Quadrant-Based MBR

We split the space using a quadrant based minimum bounding rectangle, named a Q-MBR. To construct the Q-MBR, we divide the space into quadrants, and create an MBR for the spatial objects in each quadrant. If the number of spatial objects in an MBR exceeds a split threshold, then the quadrant is recursively divided into smaller-sized sub-quadrants and MBRs are created for each sub-quadrant. Note that this partitioning method can create an MBR containing only a single spatial object. Figure 1 shows an example of the Q-MBR. The table shown in the figure is a list of Q-MBRs generated by the points on the left side of the figure. In this example, we assume that the capacity of the Q-MBR is four.

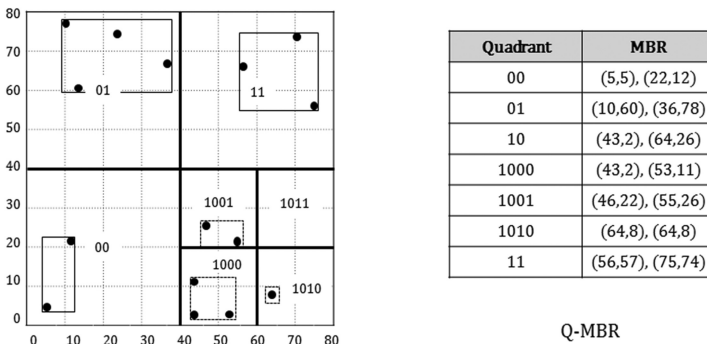


Fig. 1. An example of quadrant-based MBR

As shown in Fig. 1, Q-MBR contains both information about the quadrant and the MBR. The reason for maintaining information on both is to store the Q-MBR in the HBase table and use it as the building block of our hierarchical index structure.

The quadrant information is used as the rowkey, in order to reduce the cost of updating. If the precise MBR information is used as a rowkey, then we frequently have to create a new rowkey, because MBR information is update-sensitive. In the worst case, we create a new rowkey and redistribute every spatial object when each update occurs. Hence, the MBR information is stored in a column, which is relatively inexpensive to update. This MBR information is used for distance calculations in spatial query processing.

2.2 Hierarchical Index Structure

The spatial objects in Q-MBR are accessed in a hierarchical manner through an index tree. This index tree, named a Q-MBR tree, is implemented in an HBase table format. The structure of a Q-MBR tree is similar to that of a quad-tree. The properties of a Q-MBR tree are as follows. First, while related techniques sort spatial objects in z-order and group objects according to the auto-sharding of the table, Q-MBR can group spatial objects into smaller units as the user requires. The next property is that a Q-MBR tree does not require an additional index structure, such as a BGRP tree or the R+ tree of KR+ tree, in order to build and maintain itself. The structure of a Q-MBR tree node is described in Table 1.

Table 1. Structure of a Q-MBR tree node

Type	Component	Description
Internal node	Quadrant	The binary values of quadrant information
	MBRs of children	The coordinates of the lower left and the upper right corners of the MBR
	Number of objects	Number of objects in sub-tree
Leaf node	Quadrant	The binary values of the quadrant information
	Data objects	The list of spatial objects in this node
	Number of objects	Number of objects in this node

An internal node consists of the quadrant information of the node, the MBRs of the child nodes and the number of objects included in their sub-tree. The quadrant information of the node can be represented by binary values. When we split a node, the newly created sub-quadrants can be enumerated according to the z-order. For example, if partitioning occurs at the root node, then the sub-quadrants are named using two-bit values, such as 00, 01, 10 and 11. If the sub-quadrant is recursively partitioned, then the name of the sub-quadrant is created by concatenating the name of their parent with the newly created two-bit name.

A leaf node consists of a quadrant, a list of spatial objects, and the number of objects in this leaf node. The quadrant information and number of objects are similar to their counterparts for an internal node. The difference lies in the list of spatial objects. HBase provides a function of data filtering in order to only transmit data of interest to a

client. We define the filtering function as computing the distance between a query point and a spatial object. Therefore, the list of spatial objects contains their coordinates and ids. Figure 2 shows an example of a hierarchical Q-MBR index structure for spatial objects shown in Fig. 1. In this example, we assume that capacity of a leaf node is four.

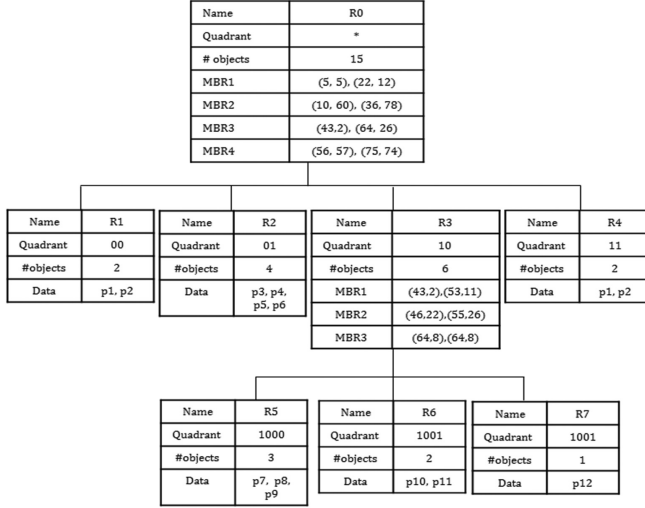


Fig. 2. An example of a Q-MBR tree

2.3 Representation of a Q-MBR Tree in HBase

In order to store a Q-MBR tree in an HBase table, it is necessary to design a schema that supports effective I/O considering the characteristics of HBase. In particular, because leaf nodes storing a group of spatial object have a large number of entries, designing table schema for efficiently loading leaf nodes from the table is important to improve the overall performance of index traversing. Due to the flexibility in schemas of HBase, a table of HBase can take one of two forms: tall-narrow and flat-wide. A tall-narrow table has a large number of rows with few columns, and a flat-wide table consists of a small number of rows with many columns.

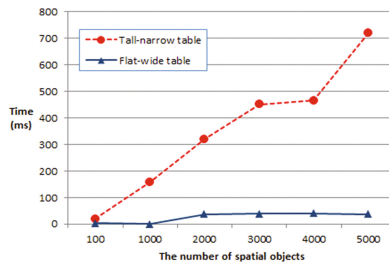


Fig. 3. Response time for loading spatial objects from an HBase table

The format of wide table is more appropriate for loading a large-sized leaf node from the HBase table. Because the spatial objects stored in a single row have the same rowkey, the time required for data fetching from a wide table is shorter. Figure 3 shows the response times for loading spatial objects from a tall-table and a wide-table. The x-axis of the graph indicates the number of spatial objects loaded from the HBase table at each time. In the case of the tall-narrow table, a desired number of rows are read at a time through a *scan* operation. In the flat-wide table, the number of spatial objects read at a time is stored in a single row, and these are obtained through a *get* operation. As shown in the figure, loading objects from the wide table delivers a better performance. Based on this observation, we store the spatial objects in each leaf node in a single row, and add a new column entry whenever a spatial object is inserted.

Row key (Quadrant)	Meta family		MBR family				Data Family				
	Is Leaf (Boolean)	# objects	00	01	10	11	d1	d2	d3	d4	
Root	0	15	(5.5). (22.12)	(10.60). (36.78)	(43.2). (64.26)	(56.57). (75.74)					
00	1	2					P1	P2			
01	1	4					P3	P4			P5
10	0	6	(43.2). (53.11)	(46.22). (55.26)	(64.8). (64.8)						
1000	1	3					P7	P8	P9		
1001	1	2					P10	P11			
1010	1	1					P12				
11	1	3					P13	P14	P15		

Fig. 4. Table for a Q-MBR tree node

Figure 4 presents the table of the Q-MBR tree for the example in Fig. 2. An internal node, such as the root or 10, has a column family of MBRs that indicates the MBR information of its children. On the other hand, leaf nodes contain a column family of data objects, which maintains a list of spatial objects. For the purpose of illustration, the column qualifier of each spatial object is enumerated from d1 to d4. However, in order to use column filtering, each spatial object should have a unique column qualifier consisting of their coordinate values. The splitting threshold of a leaf node is determined according to the batch size of the RPC in the HBase system. The batch size is the unit size of a transmission in the HBase system. This can be defined according to the requirements of the user.

3 Algorithms of Spatial Data Insertion and Range Query Processing

3.1 Insertion Algorithm for a Q-MBR Tree

Algorithm 1 presents the insertion algorithm for a Q-MBR tree. The algorithm inserts a spatial object into the leaf node whose quadrant covers the location of the spatial object. To find the appropriate leaf node, the algorithm retrieves the Q-MBR tree using the quadrant information for each node. This searching process is described in lines 2 to 7. The MBR information of children and the number of spatial objects are updated when the internal nodes are traversed. After updating the information of the current traversed node, the algorithm calculates the rowkey of the next node, and loads this from the table for the next iteration. If the appropriate leaf node is found, then the spatial object is added to the *dataFamily* of the leaf node. When the number of spatial objects in a leaf node exceeds the split threshold, the function `SplitNode()` is called to split the leaf node. The function `SplitNode(node)` returns an internal node that is the result of partitioning. The splitting process creates new children by dividing the quadrant into four sub-quadrants, and redistributes the spatial objects into newly created leaf nodes.

Algorithm 1. Insert data point for a Q-MBR tree

Input : Spatial object p , split threshold S

Output : The updated Q-MBR tree after insertion

```

node  $N \leftarrow$  root node of the Q-MBR tree
while ( $N$  is not a leaf node)
    For (each child  $C$  of  $N$ ) do
        if (quadrant of  $C$  covers  $p$ )
            update MBR of  $C$  in  $N$ 
            increase object counter of  $N$ 
         $N \leftarrow C$ 
if (object counter of  $N < S$ ) then
    add an spatial object  $p$  to DataFamily of  $N$ 
    increase object counter of  $N$ 
else if (size of  $N \geq S$ ) then
     $N = \text{SplitNode}(N)$ 
    for (each child  $C$  of  $N$ ) do
        if (quadrant of  $C$  covers  $p$ )
            update MBR of  $C$  in  $N$ 
            add an spatial object  $p$  to DataFamily of  $C$ 
            increase object counter of  $C$ 
End
    
```

Figure 5 present an example of insertion. Suppose that the spatial object p_1 , marked with a star in the figure, is inserted in the Q-MBR tree from Fig. 5(a). The algorithm starts with an examination of the root node R_0 . Because the quadrant of R_2

covers the location of p_1 , the algorithm updates the MBR of R2 in the root node, and loads R2 from the index table. The next step is to insert the object p_1 into R2. However, the number of objects in R2 exceeds the split threshold after this insertion. Therefore, R2 is split into sub-quadrants, and the spatial objects in R2 are redistributed to the children. As a result, the new leaf nodes R8, R9 and R10 are inserted into the index table, as shown in Fig. 5(b).

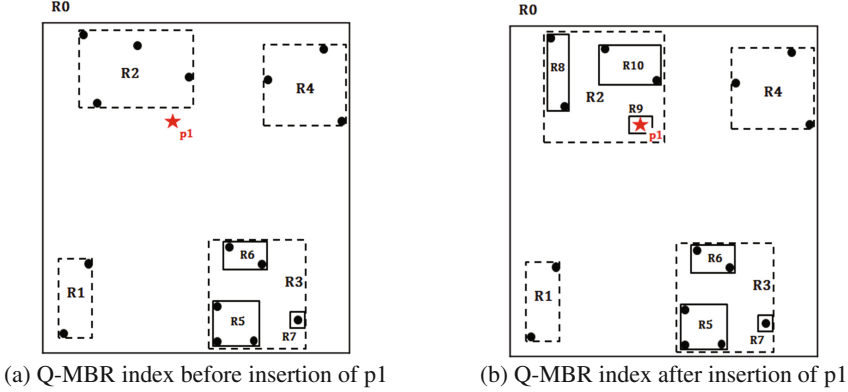


Fig. 5. An example of insertion algorithm

3.2 Range Query Algorithm for a Q-MBR Tree

The range query receives a query point q and query radius r as input, and returns a set of data points whose distance from the query point is less than r . Our algorithm for processing a range query is presented in Algorithm 2. The proposed algorithm explores the Q-MBR tree in BFS (breadth-first-search) order, and reads as many rows from the index table as possible at each time, in order to reduce the number of data requests to the region server. Two sets, named Nt and Rk in the algorithm, are maintained for this processing. The first, Nt , stores the nodes that are required to be traversed in the current iteration. Rk is a set of rowkeys to be loaded from the index table for the next iteration of the algorithm. The algorithm is terminated if there are no more nodes in either of the sets.

The algorithm starts by inserting rowkey of the root node into Rk , and loading it from the index table. If the current traversed node is an internal node, then the algorithm calculates the minimum distance between the MBRs of its children and the query point q . The rowkeys of the child nodes with distance less than the query radius r are inserted into Rk . After all of the nodes in Nt have been traversed, the algorithm loads nodes from Rk from the index table, and stores the result into Nt for the next iteration. When the current traversed node is a leaf node, the algorithm calculates the distance between the spatial objects and the query point in order to answer the query. If the distance between a spatial object p and the query point q is less than or equal to the query radius r , then the algorithm inserts p into the result set R .

Algorithm 2. Range query algorithm

Input : Query point q and range r
Output : A set of spatial objects within a given query range

```

 $Nt \leftarrow \emptyset$ 

 $Rk \leftarrow \emptyset$ 

Result set  $R \leftarrow \emptyset$ 
insert rowkey of root node into  $Rk$ 
while  $Rk$  is not empty do
     $Nt = Nt \cup \{\text{Prefetch nodes in } Rk\}$ 
    while  $Nt$  is not empty do
        for (each node  $n \in Nt$ ) do
            if ( $n$  is a leaf node) then
                for (each object  $p \in n$ ) do
                    if ( $\text{distance}(p, q) \leq r$ ) then
                         $R = R \cup \{p\}$ 
            else if ( $n$  is not a leaf node) then
                for (each child  $c$  of  $n$ ) do
                    if ( $\text{overlap}(\text{MBR of } c, q, r)$ ) then
                        insert rowkey of  $c$  into  $Rk$ 
Return  $R$ 
    
```

Figure 6 shows an example of a range query. The algorithm starts by inserting the rowkey of R_0 into Rk and loading it into Nt . There are two children, R_2 and R_3 , which overlap with the query range. Therefore, the rowkeys of R_2 and R_3 are inserted into Rk at the first iteration, and loaded together from the index table. Similarly, the rowkeys of R_9 and R_6 are inserted and loaded at the second iteration. Because R_9 and R_6 are leaf nodes, the next loop inspects the data objects of R_9 and R_6 in order to answer the query. As a result, the result set R contains the two points, p_1 and p_2 , and the algorithm is terminated, because there are no more nodes to traverse.

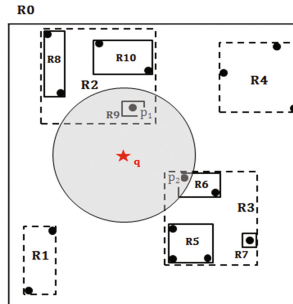


Fig. 6. An example of a range query

4 Performance Analysis

4.1 Experimental Setup and Datasets

We use synthetically generated databases, to control the size and distribution of the data. The first synthetic database contains two-dimensional uniformly distributed data, and the second contains two-dimensional data that follows a normal distribution. We implemented our method on a pseudo-distributed HBase cluster of four nodes, using HBase 0.98.0 and Hadoop 2.4.0 as the underlying system. Our experiments were performed on a physical machine that consists of a 2.5 GHz quad-core, 32 GB memory, and a 1 TB HDD, and runs 64bit Linux.

For all of the experiments, we compare our method (labeled as Q-MBR tree in the graphs) with MD-HBase [8] (labeled as MD-HBase in the graphs), and KR+ tree [9]. The average response time of 100 random queries is used for comparison. We set the parameters of MD-HBase and KR+ tree according to the analysis of [9]. MD-HBase must determine the capacity of the grid cell to group spatial objects. We set the threshold to 2500. The parameters of KR+ tree consist of the lower and upper bounds of the rectangle, and the order of the grid. We set the boundary of rectangle to (100, 50), and the order to eight. Q-MBR tree also uses the capacity of the Q-MBR as the parameter. In varying the capacity, there is a trade-off between the complexity of the index and the selectivity. We set the capacity of Q-MBR to 1024 after measuring the performance of a range query for one million datasets.

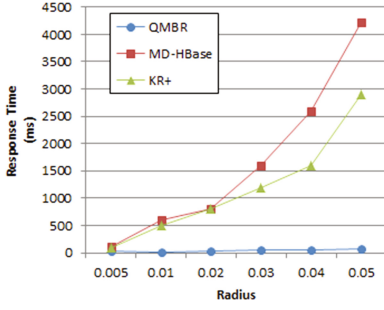
4.2 Performance Evaluation for Range Query

Effect of Query Radius

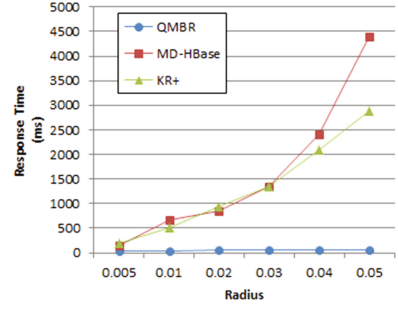
For these experiments, the database size was fixed at 10 million. Figure 7 plots the response times of range queries with a query radius increasing from 0.5% to 5% of the space. As shown in Fig. 7, Q-MBR tree outperforms MD-HBase and KR+ tree. In particular, when the size of the retrieved data increases, Q-MBR tree achieves a better performance than the other two methods because the time for loading data objects from the table is shorter. Although the table structure of KR+ tree is similar to that of Q-MBR tree, the performance of KR+ tree is inferior to that of Q-MBR tree. The reason for this is that the range query algorithm of KR+ tree is based on a key table produced by grid partitioning.

Effect of Database Size

For this set of experiments, the database size was increased from one million to 10 million points. The query radius was set as constant. Figure 8 shows that as the database size increases, the response time also increases for all methods. However, as can be seen from the figure, the rate of this increase in the response time of the Q-MBR tree is lower than for the other two methods. Because the three parameters required by KR+ tree are sensitive to the data distribution, this method shows the worst performance in this experiment.

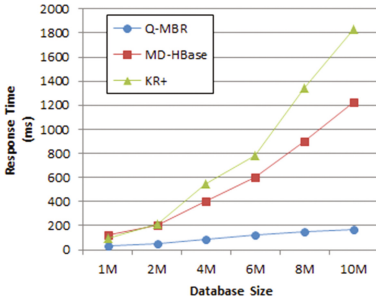


(a) Dataset with uniform distribution

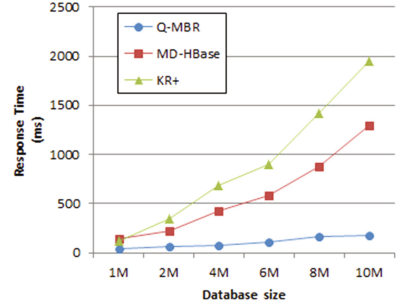


(b) Dataset with normal distribution

Fig. 7. Effect of query radius on the response time



(a) Dataset with uniform distribution



(b) Dataset with normal distribution

Fig. 8. Effect of database size on the response time

5 Conclusion

In this paper, we have presented Q-MBR tree, an efficient index scheme for handling large scale spatial data on an HBase system. The proposed scheme recursively divides the space into quadrants, and creates MBRs in each quadrant in order to construct a hierarchical index. Q-MBR provides better filtering power for processing spatial queries than existing schemes. A Q-MBR tree is stored in a flat-wide table, in order to enhance the performance of index traversal. Algorithms for range queries using Q-MBR tree have also been presented in this paper. Our proposed algorithms significantly reduce the query execution times, by prefetching the necessary index nodes into memory while traversing the Q-MBR tree. Experimental results demonstrate that our proposed algorithms outperform those of the existing two methods, MD-HBase and KR + tree. We are currently developing an effective kNN query algorithm suitable for Q-MBR tree.

References

1. Cary, A., Sun, Z., Hristidis, V., Rish, N.: Experiences on processing spatial data with MapReduce. In: SSDBM, Lecture Notes in Computer Science Scientific and Statistical Database Management, pp. 302–319 (2009)
2. Wang, K., Han, J., Tu, B., Dai, J., Zhou, W., Song, X.: Accelerating spatial data processing with MapReduce. In: IEEE 16th International Conference on Parallel and Distributed Systems, pp. 229–236 (2010)
3. Zhang, S., Han, J., Liu, Z., Wang, K., Feng, S.: Spatial queries evaluation with MapReduce. In: Eighth International Conference on Grid and Cooperative Computing, pp. 287–292 (2009)
4. Eldawy, A., Mokbel, M.F.: SpatialHadoop: a MapReduce framework for spatial data. In: 2015 IEEE 31st International Conference on Data Engineering, pp. 1352–1363 (2015)
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 Endowment* **6** (11), 1009–1020 (2013)
6. Carlini, E., Lulli, A., Ricci, L.: Dragon: multidimensional range queries on distributed aggregation trees. *Future Gener. Comput. Syst.* **55**, 101–115 (2016)
7. Li, J.F., Chen, S.P., Duan, L.M., Niu, L.: A PR-quadtrees based multi-dimensional indexing for complex query in a cloud system. In: Cluster Computing, pp. 1–12 (2017)
8. Nishimura, S., Das, S., Agrawal, D., Abbadi, A.E.: MD-HBase: design and implementation of an elastic data infrastructure for cloud-scale location services. *Distrib. Parallel Databases* **31**(2), 289–319 (2012)
9. Van, L.H., Takasu, A.: An efficient distributed index for geospatial databases. In: Lecture Notes in Computer Science Database and Expert Systems Applications, pp. 28–42 (2015)
10. Wei, L., Hsu, Y., Peng, W., Lee, W.: Indexing spatial data in cloud data managements. *Pervasive Mob. Comput.* **15**, 48–61 (2014)

Proceedings of the 7th International Conference on
Emerging Databases

Technologies, Applications, and Theory

Lee, W.; Choi, W.; Jung, S.; Song, M. (Eds.)

2018, X, 338 p. 157 illus., Hardcover

ISBN: 978-981-10-6519-4