PAPER Special Section on Data Engineering and Information Management

G-HBase: A High Performance Geographical Database Based on HBase*

Hong Van LE[†a)], ***Nonmember and*** **Atsuhiro TAKASU**[†,††], ***Member***

SUMMARY With the recent explosion of geographic data generated by smartphones, sensors, and satellites, a data storage that can handle the massive volume of data and support high-computational spatial queries is becoming essential. Although key-value stores efficiently handle large-scale data, they are not equipped with effective functions for supporting geographic data. To solve this problem, in this paper, we present G-HBase, a high-performance geographical database based on HBase, a standard key-value store. To index geographic data, we first use Geohash as the rowkey in HBase. Then, we present a novel partitioning method, namely binary Geohash rectangle partitioning, to support spatial queries. Our extensive experiments on real datasets have demonstrated an improved performance with k nearest neighbors and range query in G-HBase when compared with SpatialHadoop, a state-of-the-art framework with native support for spatial data. We also observed that performance of spatial join in G-HBase is on par with SpatialHadoop and outperforms SJMR algorithm in HBase.

key words: *HBase, high performance, spatial index, GeoHash, BGR Partitioning*

1. Introduction

Recent and rapid improvement of positioning technologies such as the Global Positioning System (GPS), sensors, and wireless networks has resulted in an explosion of geographic data generated by users. McKinsey Global Institute says that the pool of personal location data was at the level of 1 PB in 2009 and is growing at a rate of 20% per year [1]. Many systems, in both scientific research and daily life, have taken advantage of the collected location data. 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. Such systems need a database of geographic data, that can store a massive volume of data and provide high-performance spatial queries.

To meet these requirements, systems need a database management system (DBMS) that has good scalability while guaranteeing satisfactory performance with low-latency spatial queries. Although some frameworks based on MapReduce [2] such as SpatialHadoop [3], Hadoop-GIS [4] are reasonable choices for handling large volumes of spatial data, MapReduce is built for batch processing and lack of the ability to access data randomly to provide real-time query processing [5], [6]. Recently, key–value stores such as HBase [5], [6], with their scalability, fault tolerance, and random read/write capability, have shown promise. However, they do not have native support for geographic data and spatial queries, which is required in a database of geographic data.

Some methods [7], [8] use Geohash**, a linearization technique that transforms two-dimensional spatial data into a one-dimensional hashcode, to handle spatial data in HBase. However, there are two major limitations of Geohash which they do not address. The first limitation is *data skew problem*, which lead to load imbalance of parallel tasks in a distributed system and increase response time. Secondly, although Geohash uses Z-order to maintain proximity of geographic data, there are still some points that are close geographically, for example, *7* and *e* in Fig. 2, but have widely separated Geohash codes. This *Z-order problem* would cause false-positives in range query, and far apart objects are included in the same partition when partitioning data using Geohash. Besides, it is necessary to design spatial queries to adapt to the spatial indexes and the HBase structure.

We have developed G-HBase - a high-performance geographical database based on HBase. The goal of the system is to provide a distributed and efficient system that can handle a large volume of data, and support computationally expensive spatial queries. First, we used Geohash as rowkeys in HBase to distribute large datasets across a multi-server cluster. Then we proposed a novel partitioning method, the binary Geohash rectangle partitioning (BGR Partitioning), to partition tables in HBase based on the data density. We also designed algorithms to process various intensive spatial queries based on the designed index structure and proposed partitioning method. Main contributions are:

- We provide a distributed database for geographic data based on HBase which can store a large volume of data and support high-performance spatial queries.
- We propose a novel partitioning method named BGR Partitioning to handle limitations of Geohash and improve the performance of spatial queries.
- Extending on the original version of the paper [9]:

Manuscript received June 28, 2017.
Manuscript revised November 6, 2017.
Manuscript publicized January 18, 2018.

[†]The authors are with the Dept. of Informatics, SOKENDAI (The Graduate University for Advanced Studies), Tokyo, 101–8430 Japan.
[††]The author is with the National Institute of Informatics, Tokyo, 101–8430 Japan.

*This paper is an extended version of the work presented at 26th International Conference on Database and Expert Systems Applications - DEXA 2015

a) E-mail: l-van@nii.ac.jp

DOI: 10.1587/transinf.2017DAP0017

**http://geohash.org

– We introduce an algorithm to process spatial join using MapReduce on HBase. To the best of our knowledge, G-HBase is the first geographical database that supports spatial join query on HBase (Sect. 5.3, Sect. 6.5).
– We also present an algorithm using range bounding geohashes to process range query that can avoid Z-order problem (Sect. 5.2, Sect. 6.4).

2. Related Work

When considering scalable data processing systems for large datasets, systems based on MapReduce [2] have dominated. Many researchers have studied on processing spatial queries using MapReduce such as [10]–[13]. Also, there are some proposed systems including SpatialHadoop [3], [14], an spatial extension of Hadoop [15], Hadoop-GIS [4], an Hive [16] integrated spatial data processing system, SECONDO [17], and ESRI Hadoop [18] that are based on MapReduce for processing large volumes of spatial data. Some of them utilize R-Tree [19] family such as R+-Tree [20], R*-Tree [21], STR [22] to efficiently handle spatial data. Though, such systems are suitable in the context of batch processing; they still have a high latency compared with the requirements of a low-latency system.

Spatial support has also been extended to NoSQL-based solutions. [7], [8] adopt Geohash to handle spatial data in HBase. HGrid [23] builds a hybrid index structure, combining a quad-tree and a grid as primary and secondary indices in HBase. Nonetheless, these systems did not concern with spatial join query, which is also an important and expensive spatial query. GeoMesa [24] introduces a spatiotemporal index structure based on Geohash on top of Apache Accumulo[†]. It interleaves Geohash and timestamps into the design index and achieves acceptable results when storing data at 35-bit resolution. The problem is different from G-HBase as they propose an index structure for spatio-temporal data, while G-HBase focuses on geoghraphical data. Even though they also utilize Geohash, they did not address the data skew and Z-Order problems of Geohash.

MD-HBase [25] layers a multidimensional index over the key-value store by using Z-ordering [26] as linearization technique and utilizes multidimensional index structures, such as the K-d tree [27] and the Quadtree [28] to solve the Z-Order limitation on top of HBase. However, MD-HBase physically splits regions in HBase based on k-d tree or quadtree, so it requires effort to modify HBase split policy to distribute data into regions. The modification may lead to an inconsistent state when an analysis query concurrent to a split. Our index requires neither additional modification in HBase base code nor concern about consistency while splitting regions, therefore it can be utilized on top of not only HBase but also other key-value stores such as Accumulo, Cassandra. Furthermore, because our BGR partitioning does not require physical partition, it can be applied to a spatial join query with multiple tables and only costs minimal, whereas MD-HBase need further study about how to apply its index to process such query.

There are numerous research efforts to develop both in-memory [29]–[31] and disk-based [32]–[36] spatial join algorithms. In the past decade, with the explosion of collected data, many studies have been proposed to process spatial join in parallel. [37] presents two parallel partition-based spatial-merge join algorithms (PPBSM) based on the partition-based spatial-merge join (PBSM) [35], a classic spatial join algorithm which partitions input records according to a uniform grid. PPBSM reduces the effects of skewed data distribution by declustering space into a number of grid tiles, then evenly assigning tiles into nodes in the cluster. Recently, with the emergence of MapReduce, Spatial Join with MapReduce (SJMR) [38] has been proposed to adopt PPBSM algorithm into MapReduce. Though SJMR helps to distribute tiles in dense area into different partitions, it does not map the tiles based on data density, so still has imbalance problem with skewed data. In G-HBase, instead of using grid partition, we utilized the proposed BGR Partitioning to split the space based on data density of the two input tables to get the better load balance in parallel processing.

3. Overview of G-HBase

G-HBase is based on Apache HBase, a distributed scalable database which is modeled after Google's BigTable [39]. Figure 1 gives the overview of G-HBase architecture. G-HBase includes two components, namely, *indexes* component and *query processing* component.

The *indexes* component includes the index structure (Geohash) and the partitioning method (BGR Partitioning). Input data including points and non-point objects data are inserted using Geohash to encode. After data insertion, the table is partitioned based on the data density using BGR Partitioning method. Depending on the type of query, the *query processing* component will decide to use the BGR Partitioning method or not and how to utilize the method to achieve the best performance.

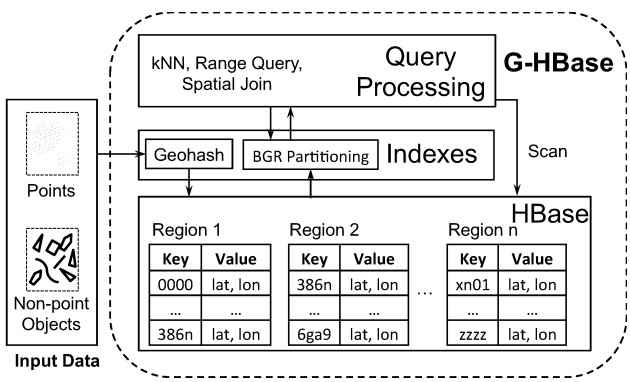


Fig. 1 Architecture Overview of G-HBase

[†]http://accumulo.apache.org

4. Indexes Component

4.1 Geohash as Rowkeys

The first part of the *indexes* component is Geohash. HBase is a key-value store (KVS) in which data are stored as a list of key–value pairs sorted by uniquely identified rowkeys. Therefore designing the unique rowkeys is one of the most important aspects of HBase schema. However, geographic 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.

Geohash is a hierarchical spatial data structure that subdivides space into uniform rectangles, then uses Z-order (Fig. 2) to encode rectangles. To best take advantage of Geohash, we store the spatial data in HBase where the rowkeys are geohashes. HBase stores data in lexicographical order of rowkeys, thus objects that are nearby geographically will also be close to each other in the database. Geohashes having a common prefix are always close to each other geographically, enabling us to do proximity search using a prefix filter.

4.1.1 Storing Points

Figure 3 shows three steps to generate a geohash for a point. In the first step, longitude and latitude dimensions are represented by longitude and latitude trees, respectively. The root node has the bound S, the maximum range of the corresponding dimension (for example, S_x is 360, S_y is 180). Geohash recursively divides nodes in the trees into two equal-size children, and then assigns bit 0 (resp. 1) if the location is in the left (resp. right) child. Then, Geohash interleaves bits from the two dimensions to generate a bit sequence. Finally, each five bits in the bit sequence is encoded into a character in Base32.

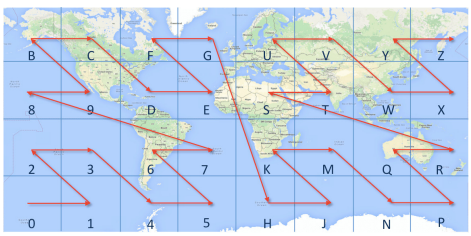


Fig. 2 Z-order traversal with 1-character geohashes

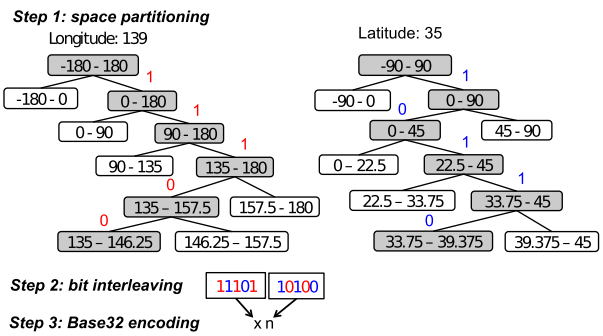


Fig. 3 Generation of a two-character geohash for a point

4.1.2 Storing Non-Point Objects

There are two approaches to index a non-point object using Geohash. First, single-assignment approach uses only one geohash which covers the whole object. Second, multiple-assignment approach decomposes a non-point object into a list of disjoint geohashes that cover the non-point object. The drawback of the multiple-assignment is that objects are duplicated, which leads to increase of data redundancy and space requirement. Whereas, single-assignment approach can minimize the data redundancy. However, it leads to wasted overlapped area and more false positives during query processing. For example, if the object in Fig. 4 is indexed using single-assignment approach, its rowkey is *w*. The object only overlaps four geohashes *ww, wx, wy, wz*, therefore 28 geohashes from *w0* to *wv* are wasted overlapped area. If a query scans the wasted overlapped area, e.g. *w0*, the object is obtained even though it does not overlap the area *w0*.

There is trade-off between data redundancy and the wasted overlapped area to determine which approach to store non-point objects. In the study [32] about redundancy in spatial databases using indexes based on z-order, it is said that small amounts of redundancy (between 30% and 70% depending on the dataset) provide the best results. Therefore, in this work, we use multiple-assignment approach, but minimize the data redundancy.

To minimize data redundancy, we first use single-assignment approach by calculating a *candidate* – the smallest geohash which contains the centroid of the minimum bounding rectangle (MBR) of the object, and both edges are longer than the edges of the MBR. If the *candidate* covers the whole object, it is indexed using one geohash. Otherwise, we can decrement the length of the *candidate* to achieve a geohash which covers the whole object. However, this will cause a large wasted overlapped area. Therefore, instead, all eight adjacent geohashes of the *candidate* are checked if they intersect the object. An extended study from [32] about redundancy in spatial databases based on

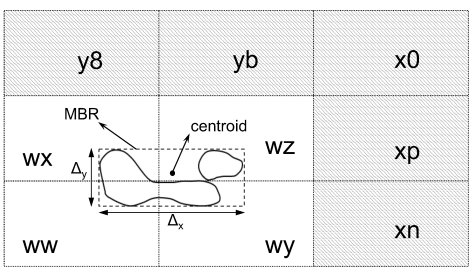


Fig. 4 Bounding geohashes for a non-point object

key-value stores is needed. We leave such extension as future work.

A geohash primarily denotes a rectangle. To simplify the explanation, here we consider the longitude dimension of the rectangle. The latitude dimension is analogous. The horizontal edge of the rectangle corresponds to a node in the longitude tree. For example, the 2-character geohash *xn* in Fig. 3 represents a rectangle with the horizontal edge corresponding to the node (135-146.25) at the fifth level of the longitude tree. The edge length of the node in level n is given by:

$$\Delta = S/2^n \tag{1}$$

where S is the length of the root node.

Definition 1. Horizontal MinRangeHash

For a rectangle r, the *MinRangeHash* corresponding to the horizontal range of r is defined as the geohash g_x that satisfies:

1. g_x contains the centroid of r,
2. the horizontal edge of g_x is larger than the horizontal edge of r, and
3. g_x is the longest geohash that satisfies the conditions 1 and 2.

The vertical *MinRangeHash* g_y corresponding to the vertical range of r is analogous.

To compute the horizonal *MinRangeHash* g_x, we approximate the horizontal range to the smallest node in the longitude tree of which the node size is larger than the range. Based on [40], the level of the node can be understood as the precision of such approximation and is calculated as:

$$n = \left\lfloor \frac{ln(S/\Delta)}{ln(2)} \right\rfloor \tag{2}$$

The length l_x of the *MinRangeHash* is calculated based on the precision as in Eq. (3). Similarly, we can compute the length of its vertical *MinRangeHash* as in Eq. (4).

$$l_x = \begin{cases} 2\lfloor n_x/b \rfloor + 1, & \text{if } (n_x \mod b) = 0 \\ 2\lfloor n_x/b \rfloor + 1, & \text{if } (n_x \mod b) \geq \lfloor b/2 \rfloor \\ 2\lfloor n_x/b \rfloor, & \text{otherwise} \end{cases} \tag{3}$$

$$l_y = \begin{cases} 2\lfloor n_y/b \rfloor + 1, & \text{if } (n_y \mod b) = 0 \\ 2\lfloor n_y/b \rfloor + 1, & \text{if } (n_y \mod b) \geq \lceil b/2 \rceil \\ 2\lfloor n_y/b \rfloor, & \text{otherwise} \end{cases} \tag{4}$$

In Eq. (3) and Eq. (4), b is the number of bits to encode a character. In the case of Geohash, b is five because Geohash uses Base32 to encode.

Algorithm 1 describes the algorithm to calculate the list. To reduce the data size inflation when assigning an object to multiple geohashes, we try to make the list with the minimum number of geohashes. Therefore, we first compute a *candidate* – the smallest geohash which contains the centroid of the minimum bounding rectangle (MBR) of the object, and both edges are longer than the edges of the MBR. To determine the *candidate*, we get the *MinRangeHash* g_x and g_y of the MBR, and calculate the maximum geohash of the two *MinRangeHashes*.

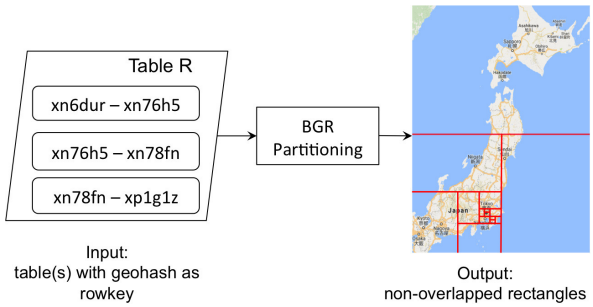


Fig. 5 Input and output of the BGR partitioning

Algorithm 1: Get a list of geohashes that cover a non-point object

input : a non-point object o
output: list of geohashes that cover the object
1: $r \longleftarrow$MBR of o
2: $g_x \longleftarrow$horizontal MinRangeHash of o
3: $g_y \longleftarrow$vertical MinRangeHash of o
4: $candidate \longleftarrow \max(g_x, g_y)$
5: $results \longleftarrow candidate$
6: **for** $ad \in candidate.getAdjacent()$ **do**
7: **if** *ad intersects with the object* **then**
8: $results \cup ad$
9: **return** *results*

After getting the *candidate*, it is added to the resultant list. Because both edges are longer than the edges of the MBR, if the centroid is close to the centroid of the *candidate*, the *candidate* may cover the object. In the case the *candidate* does not cover the object, all its eight adjacent geohashes are checked if they intersect the object then added to the result list. Finally, the object is inserted into HBase with keys corresponding to the geohashes in the list.

4.2 BGR Partitioning

The second part of the *indexes* component is BGR Partitioning. To store large tables into a distributed cluster, HBase provides an auto-sharding capability, which dynamically splits tables into a number of smaller chunks, namely regions, when the quantity of data becomes excessive. However, region split is based on one dimensional geohash code, and the Z-order problem of geohash could lead to a region may contains objects which are non-consecutive in the space. We propose BGR Partitioning method that partitions a table into disjoint rectangles in two dimensional space based on the density of the table.

In G-HBase, after data is ingested using geohash as rowkey and the table is already physically splitted into regions, a BGR partitioning logically partitions data into non-overlapped rectangles. Figure 5 shows input and output

Table 1 An example of the LCP-based segmentation

Region	Start Rowkey	End Rowkey	LCP	Rectangles
R1	**xn**6dur	**xn**76h5	xn	xn6, xn7
R2	**xn7**6h5	**xn7**8fn	xn7	xn76, xn77, xn78
R3	**x**n78fn	**x**p1g1z	x	xn, xp

of the BGR partitioning method. The input of the method could be a single table or multiple tables. The method includes two steps as in Algorithm 2. The first step is dividing regions of the table(s) into segments based on Longest Common Prefix (LCP) of geohashes. Secondly, all segments are inserted into a BGRP Tree, a temporal data structure to subdivide the segments into disjoint rectangles. The output of the method is all leaves of the BGRP Tree. Each leaf in the tree is a rectangle which can be represented by a geohash range. The output of the BGR Partitioning could be used as input of an R-Tree as in kNN query or partitions to process spatial join in parallel using MapReduce.

The range of regions in a table is updated only when region is splitted. Because the BGR partition is based on the range of regions, we only apply it when an insertion or a update causes a region split. When deleting data, the data is marked as deleted instead of being physically deleted. Therefore, the deletion in HBase does not cause any change in the range of regions and we do not need to apply BGR partitioning after each deletion.

Algorithm 2: BGR Partitioning for a single table

```
   input : a table T
   output: a list of disjoint partitions
1: tree.initialize()
2: foreach region ∈ T do
3:     S ⟵LCPSegment(region)
4:     for s ∈ S do
5:         insert(tree, s)
6: return tree.allLeaves()
```

The first step is LCP-based segmentation. For each region, we find the LCP of the start and end rowkeys, then obtain the character next to the LCP from the start rowkey (c_1) and the end rowkey (c_2). Geohash uses Base32 to encode spatial points, so we concatenate the LCP with each character from c_1 to c_2 in Base32 to create a new geohash. By using this algorithm, we can ensure that the list of segments will cover all points in the region.

Table 1 shows an example of the LCP-based segmentation. In this example, we have three regions in the geohash range from *xn6dur* to *xp1g1z*. The result of the segmentation is a list of rectangles that cover all points in the three regions. In this example, we can assume that the areas involving rectangles with longer geohashes, i.e., *xn76*, *xn77* and *xn78*, have the higher density than other areas.

In the example in Table 1, some rectangles contain others, such as *xn*, *xn7* and *xn77*. To partition the table into non-overlapped partitions, we do further partition using a BGRP Tree. The BGRP Tree is the core of BGR Partitioning. We build the tree satisfies the following properties.

- Each node (rectangle) is within a Geohash range
- 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, so we can use the BGRP Tree with multiple tables.

Figure 6 shows the insertion of all segments in Table 1 into the BGRP Tree. Algorithm 3 describes the algorithm to insert a segment into the BGRP Tree. We first do the search which descends from the root, recursively searching subtrees for the highest-level node that contains the input segment. If a node does not contain the segment, we do not search inside that node.

Algorithm 3: Insert a segment to a BGRP Tree

```
   input : a tree t, a segment s
1: if t is empty then
2:     t.setRoot(s)
3:     return
4: n ⟵search(t, s)
5: if n is null then
6:     newRoot ⟵getLCP(root, s)
7:     newRoot.addChild(root)
8:     newRoot.addChild(s)
9:     t.setRoot(newRoot)
10:    return
11: if n is leaf then
12:    BGRPTask(n, s)
13: else
14:    if n = s then
15:        for c ∈ n.getChildren() do
16:            BGRPTask(n, c)
17:    else
18:        n.addChild(s)
```

After finding the containing node, the segment is inserted into the node. A BGRP task is a binary partition task which is necessary for some cases in the insertion. 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 is a set of subrectangles that includes the smaller rectangle and covers all the surface of the bigger one.

Each subrectangle in the output need to correspond to a geohash range. As explained in Sect. 4.1, geohash is generated by binary partitioning, so if we divide a geohash into two equal rectangles, each rectangle will correspond to a geohash range. Therefore, the BGRP task 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 [41]. Figure 7 describes the BGRP task for the inputs *xn* and *xn6*. For a rectangle whose geohash range length is r, the task will require $log_2 r$ binary partition steps

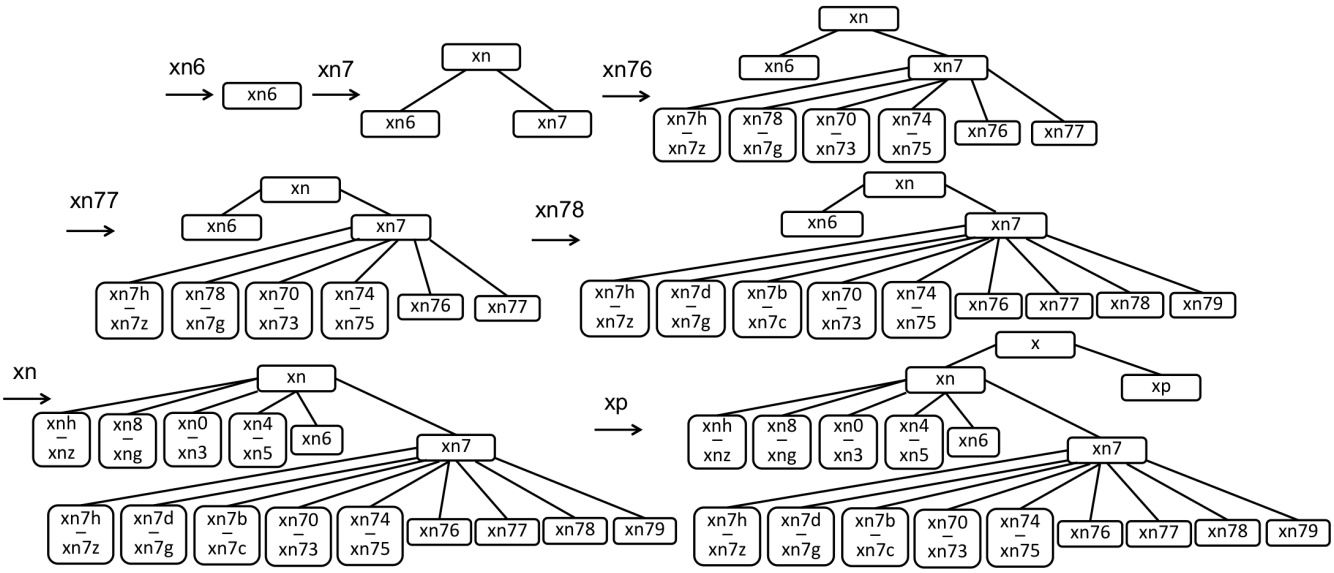


Fig. 6 Insertion in BGRP Tree

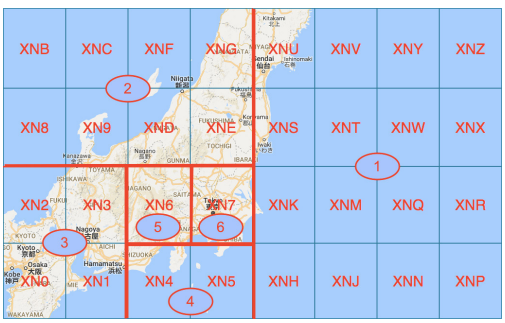


Fig. 7 The BGRP task

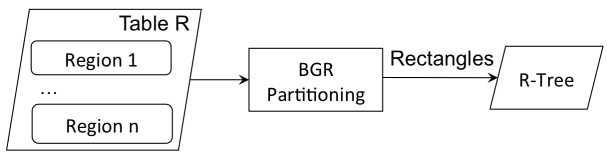


Fig. 8 The index to process kNN query

and will result in a set of $log_2 r + 1$ subrectangles.

We execute a BGRP task when inserting into a leaf, as shown for the insertion of *xn76* and *xn78* in Fig. 6. The BGRP task is also applied when the insert segment exists in the tree as a nonleaf node. Insertion of *xn* in Fig. 6 is an example of this case. Before being inserted, *xn* was a nonleaf node with only two children, *xn6* and *xn7*, which do not cover the entire surface of *xn*. Therefore, we need to partition *xn* to ensure that the surface of *xn* is completely covered in leaves. After inserting all the segments into the tree, all leaves in the tree are partitions of the table.

5. Query Processing Component

Spatial query processing strategy in the *query processing* component includes two steps. *Filter step:* Using the spatial index to quickly eliminate objects that do not satisfy the query. The output of this step is a set of candidates which possibly are included in the query result. *Refinement step:* Examining all the candidates to eliminate false objects.

5.1 k Nearest Neighbors

Filter step: With 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 regions. Because points that share the same prefix are near each other in the space, the scan returns all points that are near the query point.

Inspired by Google's BigTable [39] coprocessors, HBase also supports efficient computational parallelism by using "coprocessors". By using this parallelism, in G-HBase, we execute the refinement step inside each of the regions. Processing kNN in parallel helps reducing I/O in case scan results is larger than k. However, if all the regions return results to the client, it is still heavy I/O load when there are a large number of regions. To prune unnecessary scanning, we use an R-Tree as an additional index tier as in Fig. 8.

First, data is inserted into a table using geohash as rowkey. The table is splitted into regions. Then, we apply the BGR partitioning into the table. Finally, the output of the partitioning, all leaves of the BGRP Tree, are inserted into the R-Tree. When processing kNN, we find the rectangles in the R-Tree that may contain query results, then scan only the found rectangles, thereby pruning the scanning on unrelated regions. All the steps for R-Tree creation are processed in the background, removing the need for R-

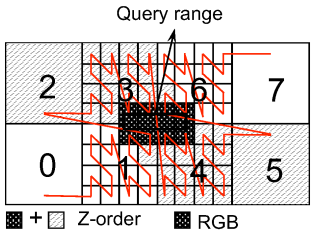


Fig. 9 Scanning area

Tree creation overhead when processing queries.

Refinement step: After obtaining the scan results, the client then refines the results to get the k nearest neighbors of the query point. If not enough nearest neighbors are found, the filter step is conducted again with shorter prefix until enough k points are found. Otherwise, we check the accuracy of the result. Same as in [3], if the distance from the point to its k^{th} furthest neighbor is shorter than the distance from the point to four edges of the prefix rectangle, the query is terminated. Otherwise, we draw a circle centered at the input point with the distance from the point to its k^{th} furthest neighbor is the radius. Then, we scan the neighbors of the prefix which overlap the circle and then get the final k nearest neighbors.

5.2 Range Query

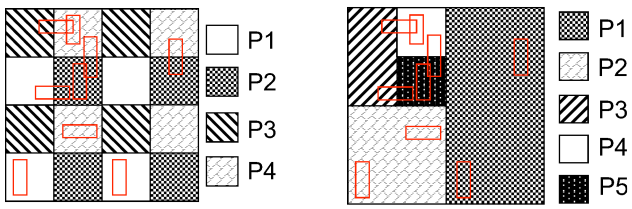
Filter step: Query range can be simply processed by scanning between start geohash and end geohash of the range, making many false positives due to Z-order traversal. For example, in Fig. 9, the query range starts with geohash $1x$ and ends with geohash 62, causing a lot of redundant scan in rectangles 2, 3, 4, 5. To reduce the false positives, we scan using range bounding geohashes (RBG) of the query range.

Algorithm 4 describes how to determine RBG of a query range. Same as Sect. 4.1.2, we calculate the *MinRangeHash* g_x and g_y corresponding to the horizontal and vertical edge of the query range, respectively. As experienced in the work [42], the imbalance between the horizontal and vertical edge in the range query would lead to higher query response time. The query range could have a wide horizontal edge much longer its vertical edge or vice versa. In such case, if we choose the maximum of g_x and g_y as in Sect. 4.1.2 as the hash length of the RBG, it would contain many points outside the query range. To reduce the redundant scan area, the algorithm starts with computing the minimum geohash g of the two *MinRangeHash*. Then g is expanded by adding all its neighbors into a list until the query range is covered by the list. With each geohash in the list, G-HBase generates a scan request with the geohash as a prefix filter. We do not further reduce the length because it leads to longer list of geohashes, causing more scan requests between the client and the cluster.

Refinement step: After getting scan results, G-HBase checks whether the query range contains points in the scanned results and eliminate all points that are not the query range. In G-HBase, refinement step in range query is executed inside each region of HBase, so that we can prune unrelated points sent from regions to the client. We named the refinement step in each region "range filter".

Algorithm 4: Get RBG for a query range

```
   input : a query range q
   output: list of geohashes that cover the query range
1: r ⟵ MBR of q
2: g_x ⟵ horizontal MinRangeHash of q
3: g_y ⟵ vertical MinRangeHash of q
4: g ⟵ min(g_x, g_y)
5: results.add(g)
6: while results do not cover r do
7:     for r ∈ results do
8:         candidates.add(r)
9:         for ad ∈ r.getAdjacent() do
10:            if ad intersects with q then
11:                candidates.add(ad)
12:    results ⟵ candidates
13: return results
```



(a) SJMR partitioning (b) BGR partitioning

Fig. 10 Partition data to parallel spatial join processing

5.3 Spatial Join

Filter step: The spatial join query is a fundamental operation in a geographical database. The inputs of the query are two spatial datasets R and S and a spatial relation such as intersection, enclosure, nearness. The output of the query is a set of all pairs (r, s) where $r \in R$, $s \in S$ and r has the spatial relation with s. The input datasets can be complex objects such as lines, polygons, multi-polygons and calculating whether two such complex objects satisfy a join condition is very computationally expensive. Furthermore, because spatial join query often involves processing the whole datasets, it is a costly operation.

SJMR [38] was proposed to process spatial join in a distributed system using the MapReduce paradigm. In map stage, it partitions data into tiles, numbers the tiles according to Z-order and maps the tiles to partitions with a round robin scheme. Though it helps to distribute tiles in a dense area into different partitions, it makes redundant tiles in a sparse area and does not map the tiles based on data density. For example, in Fig. 10 (b), SJMR partitions the space into 16 tiles and assigns them to four partitions. While the fourth partition contains five objects, the third partition has only one objects. In G-HBase, we also use the MapReduce paradigm to process spatial join query. However, to achieve

better balance in parallel processing, we present BGRP join algorithm which utilizes the BGR Partitioning to partition space based on the density of the input data.

The BGRP join algorithm includes three phases as in Algorithm 5. In *phase A*, BGR Partitioning is applied with both input tables to create a BGRP tree. We assumed that the two input tables were ingested using geohash as rowkey. As explained in Sect. 4.2, we divided each region in table R and S into segments based on LCP of start rowkey and end rowkey of the region, then inserted the segments into the BGRP tree. The BGR partitioning algorithm does not physically partitions the records in the two table. It only defines the geohash ranges to be processed in map functions. Therefore, the overhead for this step is very small as in the experiments in Sect. 6.5.

Then, *phase B* generates combined splits corresponding to leaves in the tree. For each leaf, the data in the geohash range in the two tables may be located in different machines. We scan in table R and S to get the lists R_i and S_i of objects which are in the geohash range of the leaf, respectively. A combined split is a pair of the two lists R_i and S_i. Each combined split contains the server names hosting its geohash range. To make use of data locality, the MapReduce framework checks the server names, and if MapReduce worker nodes process is running on the same machines, it will preferably run it on one of those servers. Because HBase is colocated with MapReduce on a same cluster, the scan in the server will be able to retrieve all data from the local disk. Finally, in *phase C*, inside each map function, we use the plane sweep [30], a widely used in-memory spatial join approach, to join the two lists.

Algorithm 5: BGRP Join

```
   input : table R and table S
   output: table J which contains all overlapped pair (r, s) from R
           and S
   /* Phase A: applying BGR Partitioning for both
      tables                                              */
1: tree.initialize()
2: foreach region_r ∈ R do
3:     for s_r ∈ LCPSegment(region_r) do
4:         insert(tree, s_r)
5: foreach region_s ∈ S do
6:     for s_s ∈ LCPSegment(region_s) do
7:         insert(tree, s_s)
   /* Phase B: generating combined splits                 */
8: foreach p_i ∈ tree.allLeaves() do
9:     R_i ⟵ R.scan(p_i)
10:    S_i ⟵ S.scan(p_i)
   /* Phase C: processing map function with each
      split                                               */
11: function map(p_i, pair(R_i, S_i)):
12:    results ⟵ planeSweep(R_i, S_i)
13:    for j ∈ results do
14:        if j.referencePoint ∈ p_i then
15:            J.ingest(j)
```

Refinement step: Because a non-point object may belong to multiple partitions, there are duplicated records after the filter step. We use the reference-point technique [43] to avoid duplication in the results. For each detected overlapping pair, we first compute the intersection of the pair. Then, if the reference-point (e.g., top-left corner) of the intersection is contained in the partition, the pair is reported as a final answer.

6. Experimental Results

6.1 Cluster Configuration

We built a cluster with 64 nodes. Each node had two virtual cores, 16 GB memory and a 288 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 on each datanode. Policy for region splitting was set to *ConstantSizeRegionSplitPolicy* and maximum file size of each HRegion was set to 512 MB. To conduct queries using MapReduce, we installed SpatialHadoop v2.4 and configured one master and 60 slaves. The system was configured to run one map or reduce instance on each node.

6.2 Dataset Description

We used four real datasets with points (T-Drive [44], OpenStreetMap (OSM[†] Nodes) and non-point objects (OSM Lakes, OSM Parks) in our experiments.

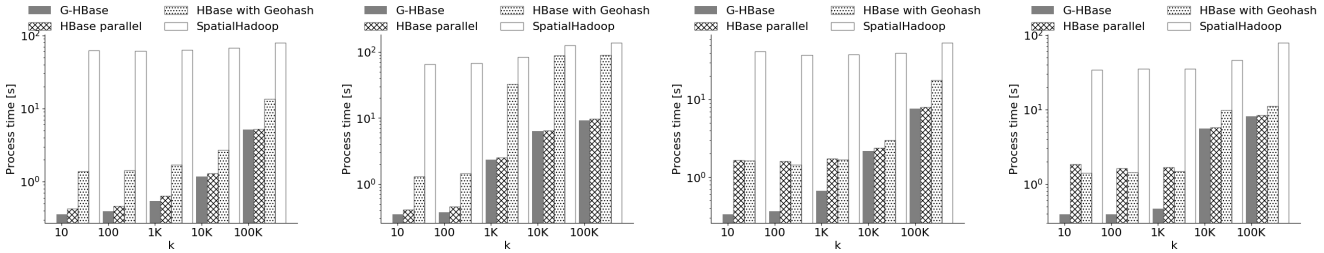
T-Drive. T-Drive was generated by 30,000 taxis in Beijing during Feb. 2 to Feb. 8, 2008 within Beijing. The total number of records in this dataset is 17,762,390. It required more than 700 MB and was split into eight regions in the HBase cluster.

OSM Nodes. OSM includes datasets of 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. We inserted the OSM Nodes dataset into 860 regions in the HBase cluster.

To evaluate kNN query and range query, we chose two groups of points for the experiments, namely, a group of high-density points and a group of low-density points. We summarized the number of points in the area of each 25-bit geohash and sorted in descending order. High-density points are the points in crowded areas, which are the first ten geohashes in the list. Conversely, low-density points are the points in uncongested areas, which are the last ten geohashes in the list.

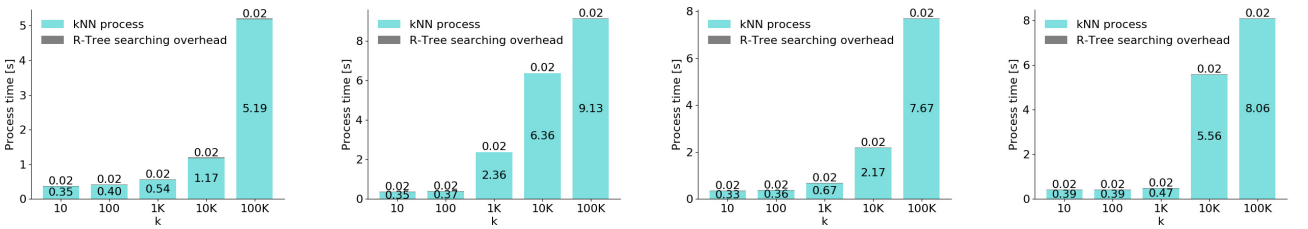
OSM Lakes and OSM Parks. We used two datasets, OSM Parks and OSM Lakes, to conduct spatial join query experiments. To evaluate the query with various data sizes, we extracted 0.01%, 0.1%, 1%, 10%, 100% of each dataset. Table 2 and Table 3 describe detail information of all extracted datasets from OSM Lakes and OSM Parks dataset,

[†]http://www.openstreetmap.org



(a) High-density in T-Drive (b) Low-density in T-Drive (c) High-density in OSM Nodes (d) Low-density in OSM Nodes

Fig. 11 Performance of kNN queries



(a) High-density in T-Drive (b) Low-density in T-Drive (c) High-density in OSM Nodes (d) Low-density in OSM Nodes

Fig. 12 R-Tree searching overhead

Table 2 OSM Lakes datasets

	0.01%	0.1%	1%	10%	100%
# of objects	841	8,419	84,193	841,931	8,419,319
File size	6 MB	56 MB	490 MB	2.5 GB	9.7 GB
# of rows	1,461	15,162	152,809	1,509,175	14,977,600
# of regions	1	1	4	18	86

Table 3 OSM Parks datasets

	0.01%	0.1%	1%	10%	100%
# of objects	996	9,961	99,618	996,188	9,961,883
File size	8 MB	58 MB	563 MB	3.1 GB	10 GB
# of rows	1,817	18,049	180,599	1,792,808	17,816,197
# of regions	1	1	4	23	86

respectively.

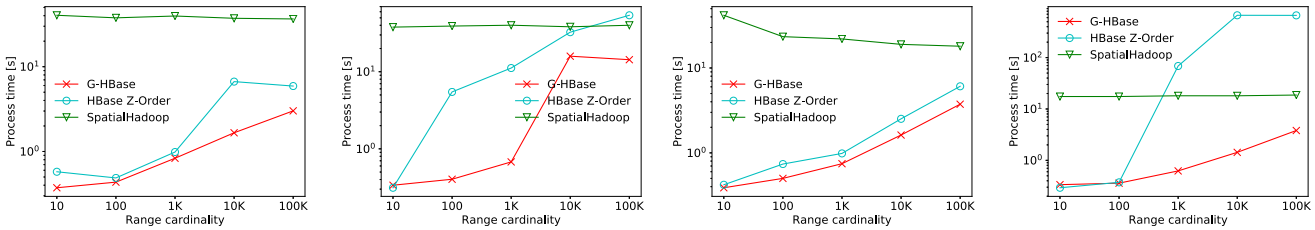
6.3 k Nearest Neighbors

We evaluated the performance of kNN queries using the two datasets, T-Drive and OSM Nodes. We conducted kNN query processing method on G-HBase and compared with the parallel query in HBase, the query using only Geohash and the query in SpatialHadoop as a baseline method. Figure 11 plots the process time in seconds for the performance of kNN queries with the T-Drive and OSM Nodes dataset. We vary k as 10, 100, 1,000, 10,000 and 100,000. The process time is measured by calculating the elapsed time between when the system starts scanning data and when all results are returned.

With both datasets, we observed that the kNN query in G-HBase outperformed all other kNN query methods. Note that response times of kNN query in G-HBase are proportional to k and about tenfold to hundredfold lower than queries using MapReduce in SpatialHadoop. Queries using MapReduce presented the poor performance regardless of the k because of the brute force parallel scan of the entire table. In G-HBase, by using R-Tree, we can define the regions that may store the results before processing kNN, then only process kNN in those regions, so we do not need to scan the whole regions. The overhead for searching on R-Tree in G-HBase is only about 20 milliseconds as in Fig. 12, whereas MapReduce processing has overhead for the startup, cleanup, shuffling and sorting tasks, which are always cost at least one or two seconds for each task.

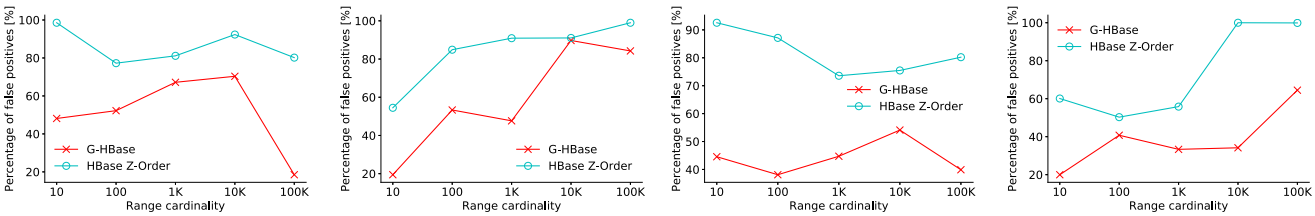
Experimental results differed in performance, depending on whether high-density or low-density points were being processed in either T-Drive or OSM Nodes dataset. With the T-Drive dataset, for low-density points and larger values of k, the queries could not find sufficient neighbors during the initial 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 in G-HBase.

R-Tree searching overhead. We also measured the overhead involved in the R-Tree searching step. Figure 12 shows the performance of kNN query in G-HBase with an R-Tree searching overhead. As shown in the figure, the cost for the R-Tree searching is around 20 milliseconds, which account for 0.21% to 5.71% of the total process time of kNN query in G-HBase. The overhead does not change much when we increase k. We only insert big rectangles representing dataset partitions instead of whole points in the dataset 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.



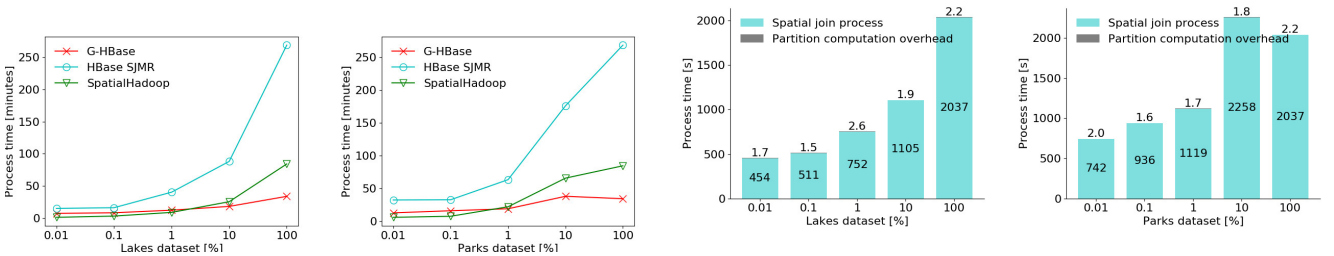
(a) High-density in T-Drive (b) Low-density in T-Drive (c) High-density in OSM Nodes (d) Low-density in OSM Nodes

Fig. 13 Performance of range queries



(a) High-density in T-Drive (b) Low-density in T-Drive (c) High-density in OSM Nodes (d) Low-density in OSM Nodes

Fig. 14 False positives in range queries



(a) With OSM Parks dataset (b) With OSM Lakes dataset (c) Overhead with OSM Parks dataset (d) Overhead with OSM Lakes dataset

Fig. 15 Performance of spatial join queries

6.4 Range Query

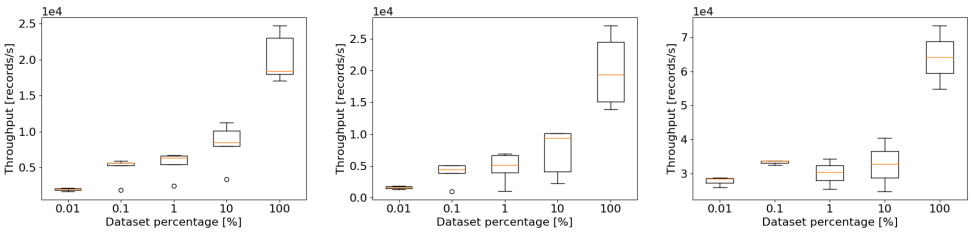
We evaluated the performance of the range query in G-HBase in comparison with SpatialHadoop and the range query using Z-order which scans between start geohash and end geohash of the range. For evaluation, we randomly sampled query ranges with the range cardinality varying from 10 to 100K. Same as the kNN query experiment, we conducted range query experiment in high-density and low-density areas.

The process times of three range query types with T-Drive and OSM Nodes datasets were plotted in Fig. 13. Query performance of G-HBase differs according to datasets and the range cardinality, but in all cases, the query in G-HBase outperformed the query in SpatialHadoop. Even though in most cases, the query using Z-order showed better performance than SpatialHadoop, it still had higher process time than in G-HBase, especially for queries in low-density areas. The main reason for the worse performance of Z-order is the false positive scans in the filter step.

Figure 14 plots the percentage of false positives of the query with Z-order and the query in G-HBase. We observed from Fig. 13 and Fig. 14 that the process times are proportional to the number of scanned points in the filter step. As illustrated in the figure, using Z-order caused more false positives in the filter step, whereas G-HBase computes the RBG to scan, therefore can improve the performance. For example, in low-density areas in OSM Nodes dataset, response times of queries using Z-order were exceptionally high. The number of scanned points in these areas is also very high with a large number of false positives, causing the decrease of query performance.

6.5 Spatial Join

We used OSM Parks and OSM Lakes datasets with various sizes as described in Sect. 6.2 to conduct spatial join experiments. Each extracted dataset from OSM Lakes dataset is joined with OSM Parks dataset and vice versa. Figure 15 exhibits performance of the BGRP join algorithm in G-HBase in comparison with SJMR in HBase and SpatialHadoop. In experiments in Fig. 15 (a), we joined the OSM Parks dataset with the extracted datasets from OSM Lakes dataset (as de-



(a) OSM Parks dataset (b) OSM Lakes dataset (c) OSM Nodes dataset

Fig. 16 Insertion throughput

scribed in Table 2). Figure 15 (b) shows results of the experiments in which we joined the OSM Lakes dataset with all the extracted datasets from OSM Parks dataset (as described in Table 3).

In both cases, we observed that using the BGRP join algorithm in G-HBase showed 2-fold to 7-fold speedup over the SJMR algorithm in HBase. SJMR gave a poor performance as it uses a uniform grid to partition, causing higher response times in some partitions which have a large number of objects to join. Meanwhile, G-HBase partitions data based on data density by using BGR Partitioning, therefore can reduce the influence of skew data. Moreover, SJMR algorithm uses both map and reduce tasks, thus costs more for shuffling and sorting between map and reduce.

SpatialHadoop also showed good performance as it uses R-Tree to deal with skew data. When joining a small dataset with a big dataset, SpatialHadoop are more efficient because it has a preprocessing step to reduce the number of partitions, then reduce the number of map tasks. In G-HBase, region partitions from the larger table created more leaves in the BGRP Tree than the smaller one, thus generating more map tasks to be processing. However, when the two input tables are both big datasets, G-HBase provided better performance. G-HBase pre-splits result table based on the leaves in the BGRP Tree, so it can distribute the insertion into multiple regions, reducing the cost of inserting a huge number of resulted objects into HBase.

BGR Partitioning overhead. We also measured the overhead of the BGR Partitioning in spatial join query (Fig. 15 (c), 15 (d)). Same as in kNN query, the cost for the BGR Partitioning is very small (around one second) in comparison with the cost for spatial join process.

6.6 Insertion Throughput

We evaluated the insert performance with both points and non-point objects. With non-point objects, we used OSM Lakes and OSM Parks datasets. With points, we used OSM Nodes dataset. Figure 16 (a), 16 (b), 16 (c) show the insertion throughput when inserting OSM Parks, OSM Lakes, OSM Nodes datasets with the number of records varied from 0.01% to 100% of the original datasets. We observed that in all cases, the more records inserted, the higher the insertion throughput. HBase dynamically split regions when the quantity of data in regions excesses a pre-defined threshold, so number of regions increases when more data are inserted. With more regions, data are inserted to multiple regions concurrently, leading to higher insertion throughput. Because generating Geohash is not computationally expensive, G-HBase can sustain a peak insertion throughput of 70,000 records per second.

7. Conclusions

We have described G-HBase, a distributed database with native support for geographic data. G-HBase is equipped various fundamental spatial queries in the *query processing* component by utilizing our proposed BGR Partitioning in the *indexes* component. Experiments with four real-world datasets including both points and non-point data demonstrated the high performance of the compute-intensive spatial queries in comparison with SpatialHadoop and other query algorithms in HBase. The results also showed that the overhead when using the proposed BGR Partitioning is a very small fraction of the process time of the queries.

In the future, we plan to support moving objects data in HBase. Moving objects databases pose many new challenges, such as querying data when its spatial relationships are continuously changing and maintaining high performance while guaranteeing high insertion throughput.

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] J.-G. Lee and M. Kang, “Geospatial big data: challenges and opportunities,” Big Data Research, vol.2, no.2, pp.74–81, 2015.
[2] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” Communications of the ACM, vol.51, no.1, pp.107–113, 2008.
[3] A. Eldawy and M.F. Mokbel, “A demonstration of spatialhadoop: An efficient mapreduce framework for spatial data,” Proceedings of the VLDB Endowment, vol.6, no.12, pp.1230–1233, 2013.
[4] 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, vol.6,

no.11, pp.1009–1020, 2013.
[5] L. George, HBase: the definitive guide, "O'Reilly Media, Inc.," 2011.
[6] N. Dimiduk, A. Khurana, M.H. Ryan, and M. Stack, HBase in action, Manning Shelter Island, 2013.
[7] K. Lee, R.K. Ganti, M. Srivatsa, and L. Liu, "Efficient spatial query processing for big data," Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp.469–472, ACM, 2014.
[8] S. Pal, I. Das, S. Majumder, A.K. Gupta, and I. Bhattacharya, "Embedding an extra layer of data compression scheme for efficient management of big-data," Information Systems Design and Intelligent Applications, pp.699–708, Springer, 2015.
[9] L.H. Van and A. Takasu, "An efficient distributed index for geospatial databases," Database and Expert Systems Applications: 26th International Conference, DEXA 2015, Valencia, Spain, Sept. 1-4, 2015, Proceedings, vol.9261, pp.28–42, Springer, 2015.
[10] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, "Experiences on processing spatial data with mapreduce," International Conference on Scientific and Statistical Database Management, vol.5566, pp.302–319, Springer, 2009.
[11] Q. Ma, B. Yang, W. Qian, and A. Zhou, "Query processing of massive trajectory data based on mapreduce," Proceedings of the first international workshop on Cloud data management, pp.9–16, ACM, 2009.
[12] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi, "Voronoi-based geospatial query processing with mapreduce," Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, pp.9–16, IEEE, 2010.
[13] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng, "Spatial queries evaluation with mapreduce," Grid and Cooperative Computing, 2009, GCC'09, Eighth International Conference on, pp.287–292, IEEE, 2009.
[14] A. Eldawy and M.F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," Proceedings of the IEEE International Conference on Data Engineering (ICDE'15), pp.1352–1363, IEEE, 2015.
[15] T. White, Hadoop: The definitive guide, "O'Reilly Media, Inc.," 2012.
[16] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," Proceedings of the VLDB Endowment, vol.2, no.2, pp.1626–1629, 2009.
[17] J. Lu and R.H. Guting, "Parallel secondo: boosting database engines with hadoop," Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on, pp.738–743, IEEE, 2012.
[18] R.T. Whitman, M.B. Park, S.M. Ambrose, and E.G. Hoel, "Spatial indexing and analytics on hadoop," Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp.73–82, ACM, 2014.
[19] A. Guttman, "R-trees: a dynamic index structure for spatial searching," Proceedings of the 1984 ACM SIGMOD international conference on Management of data – SIGMOD '84, pp.47–57, ACM, 1984.
[20] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects," 1987.
[21] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," Proceedings of the 1990 ACM SIGMOD international conference on Management of data – SIGMOD '90, pp.322–331, ACM, 1990.
[22] S.T. Leutenegger, M.A. Lopez, and J. Edgington, "Str: A simple and efficient algorithm for r-tree packing," Data Engineering, 1997, Proceedings 13th international conference on, pp.497–506, IEEE, 1997.
[23] D. Han and E. Stroulia, "Hgrid: A data model for large geospatial data sets in hbase," Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on, pp.910–917, IEEE, 2013.
[24] A. Fox, C. Eichelberger, J. Hughes, and S. Lyon, "Spatio-temporal indexing in non-relational distributed databases," Big Data, 2013 IEEE International Conference on, pp.291–299, IEEE, 2013.
[25] S. Nishimura, S. Das, D. Agrawal, and A.E. Abbadi, "Md-hbase: a scalable multi-dimensional data infrastructure for location aware services," Mobile Data Management (MDM), 2011 12th IEEE International Conference on, pp.7–16, IEEE, 2011.
[26] G.M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, International Business Machines Company, 1966.
[27] J.L. Bentley, "Multidimensional binary search trees used for associative searching," Communications of the ACM, vol.18, no.9, pp.509–517, 1975.
[28] R.A. Finkel and J.L. Bentley, "Quad trees a data structure for retrieval on composite keys," Acta informatica, vol.4, no.1, pp.1–9, 1974.
[29] P. Mishra and M.H. Eich, "Join processing in relational databases," ACM Computing Surveys (CSUR), vol.24, no.1, pp.63–113, 1992.
[30] F.P. Preparata and M. Shamos, Computational geometry: an introduction, Springer Science & Business Media, 2012.
[31] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki, "Touch: in-memory spatial join by hierarchical data-oriented partitioning," Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp.701–712, ACM, 2013.
[32] J.A. Orenstein, "Redundancy in spatial databases," ACM SIGMOD Record, vol.18, no.2, pp.295–305, ACM, 1989.
[33] M-.L. Lo and C.V. Ravishankar, "Spatial joins using seeded trees," ACM SIGMOD Record, vol.23, no.2, pp.209–220, ACM, 1994.
[34] M.-L. Lo and C.V. Ravishankar, "Spatial hash-joins," ACM SIGMOD Record, vol.25, no.2, pp.247–258, ACM, 1996.
[35] J.M. Patel and D.J. DeWitt, "Partition based spatial-merge join," ACM SIGMOD Record, vol.25, no.2, pp.259–270, ACM, 1996.
[36] N. Koudas and K.C. Sevcik, "Size separation spatial join," ACM SIGMOD Record, vol.26, no.2, pp.324–335, ACM, 1997.
[37] J.M. Patel and D.J. DeWitt, "Clone join and shadow join: two parallel spatial join algorithms," Proceedings of the 8th ACM international symposium on Advances in geographic information systems, pp.54–61, ACM, 2000.
[38] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu, "Sjmr: Parallelizing spatial join with mapreduce on clusters," Cluster Computing and Workshops, 2009, CLUSTER'09, IEEE international conference on, pp.1–8, IEEE, 2009.
[39] 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), vol.26, no.2, pp.1–26, 2008.
[40] J. Ježek and I. Kolingerová, "Stcode: The text encoding algorithm for latitude/longitude/time," Connecting a Digital Europe Through Location and Place, pp.163–177, Springer, 2014.
[41] R.A. Schumacker, B. Brand, M.G. Gilliland, and W.H. Sharp, "Study for applying computer-generated images to visual simulation," tech. rep., DTIC Document, 1969.
[42] H.V. Le and A. Takasu, "A scalable spatio-temporal data storage for intelligent transportation systems based on hbase," Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on, pp.2733–2738, IEEE, 2015.
[43] J.-P. Dittrich and B. Seeger, "Data redundancy and duplicate detection in spatial join processing," Data Engineering, 2000, Proceedings, 16th International Conference on, pp.535–546, IEEE, 2000.
[44] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang, "T-drive: driving directions based on taxi trajectories," Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems, pp.99–108, ACM, 2010.

Hong Van Le received the B.S. degrees in Information Technology from Hanoi University of Technology in 2011. She is now a PhD student at SOKENDAI (The Graduate University for Advanced Studies), Japan.

Atsuhiro Takasu received B.E., M.E. and Dr. Eng. from the University of Tokyo in 1984,1986 and 1989, respectively. He is a professor of National Institute of Informatics, Japan. His research interests are data engineering, digital libraries and data mining. He is a member of ACM, IEEE, IEICE, IPSJ and JSAI.