Pervasive and Mobile Computing 15 (2014) 48–61

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

Indexing spatial data in cloud data managements

Ling-Yin Wei [a,b,*], Ya-Ting Hsu [a], Wen-Chih Peng [a], Wang-Chien Lee [c]

[a] Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
[b] Institute of Information Science, Academia Sinica, Taipei, Taiwan
[c] Department of Computer Science and Engineering, The Penn State University, PA, USA

ARTICLE INFO

Article history:
Available online 12 July 2013

Keywords:
Cloud data management
Geographic data
Spatial index

ABSTRACT

With the proliferation of smart phones and location-based services, the amount of data with spatial information, referred to as spatial data, is dramatically increasing. Cloud computing plays an important role handling large-scale data analysis, and several cloud data managements (CDMs) have been developed for processing data. CDMs usually provide key-value storage, where each key is used to access its corresponding value. However, user-generated spatial data are usually distributed non-uniformly. In this paper, we present a novel key design based on an R^+-tree (KR^+-*index*) for retrieving skewed spatial data efficiently. In the experiments, we implement the KR^+-index on Cassandra, and study its performance using spatial data. Experiments show that the KR^+-index outperforms the-state-of-the-art methods.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

With the prevalence of the Global Positioning System and mobile devices, a large number of location-based applications, such as Foursquare and Flickr, have been developed. People can share their real-time events with friends anytime and anywhere as long as the Internet is available. For example, people can check in to a specific location and can note their activities, and they can see their friends' shared real-time information using the Foursquare application. These location-based applications induce that the amount of multi-attribute data, which at least consist of locations and time-stamps, is dramatically increasing. In order to retrieve and manage this data well, different database management systems (DBMSs) have been developed. For traditional relational database management systems (RDBMSs), there are several index structures, such as *k*-dimensional (*k*-d) trees [1], quad trees [2], and R-trees [3]. However, RDBMSs are unable to deal with thousands of millions of queries efficiently. On the other hand, distributed relational database management systems (DRDBMSs) have been developed and are able to deal with multi-attribute accesses. However, DRDBMSs are unable to maintain and retrieve data among servers efficiently, because they take much time to make sure the data is consistent by appropriately locking and updating the data.

To deal with a huge amount of data efficiently and flexibly, cloud computing is nowadays playing an important role, and new cloud data managements (CDMs), which are NoSQL databases [4], have been developed. The most prevalent NoSQL CDMs, such as HBase [5], Cassandra [6] and Amazon Simple Storage [7], are developed based on a BigTable [8] management system. Compared with DRDBMSs, these management systems have the characteristics of high scalability, high availability and fault-tolerance because they can effectively and efficiently handle a large number of data updates even if failure events occur. In addition, a BigTable management system stores data as ⟨key, value⟩ pairs, and thus these BigTable-like management systems can retrieve data efficiently by the following characteristics: (1) each ⟨key, value⟩ pair is stored

* Corresponding author at: Institute of Information Science, Academia Sinica, Taipei, Taiwan. Tel.: +886 932536416.
E-mail address: lywei.edu@gmail.com (L.-Y. Wei).

1574-1192/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.pmcj.2013.07.001

on multiple servers, and (2) each key owns multiple versions of a value. In other words, the first characteristic benefits the efficiency of retrieving data, and the second characteristic eliminates the waiting time of making data consistent. Due to the inherent restriction of a BigTable data structure, however, these management systems only support some basic operations, such as *Get*, *Set* and *Scan*. A *Get* operation retrieves values mapped by a key; a *Set* operation inserts/modifies values according to a corresponding key; a *Scan* operation returns all values mapped by a range of keys. However, these basic operations do not directly support multi-attribute accesses.

In this paper, to support efficient multi-attribute accesses of skewed data on CDMs, we propose a novel multi-dimensional index, called the KR$^+$-index,[1] on CDMs by designing Key names for leaves of an R$^+$-tree. A challenging issue is to filter out data after querying the results from a large difference of volume of data between grids. In order to describe it conveniently, we call the size of a grid the volume of the data in the grid. However, dividing a map more meticulously could reduce the differences in the grid sizes but could also reduce the efficiency of accessing data. For example, for a range query, we need to retrieve more grids for the same spatial range. According to the aforementioned observations, we expect that the differences of the grid sizes could be smaller and the time of the grid accesses could be less at the same time. Consequently, how to divide a map into grids to reach a balance between the two points plays an important role for CDMs. In this paper, we first use an R$^+$-tree [10] to divide the data, and the rectangles in the leave nodes of the tree index are treated as dynamic grids. The reasons for using an R$^+$-tree are described as follows. First, we could get a balance between the grid sizes and the times of grid accesses by adjusting the two parameters, M and m, of the R$^+$-tree. Second, compared with other variants of the R-tree, the leaf nodes do not overlap each other, and thus it is a benefit as there is no redundant retrieval of the same data from different keys and it is easy to define different keys for each rectangle of a leaf node. Moreover, the second challenge is how to design the key names of these grids to support efficient queries on BigTable management systems. We observed the characteristics of CDMs as follows: a CDM has a fast key-value search and it is fast to *Scan* keys which are ordered by a dictionary order. Based on these characteristics, we propose an approach to define the key name of a grid to support efficient queries. In addition, in this paper, we present how to deal with data insertion and deletion while using the proposed index method. We provide a guideline for setting proper parameters used in the proposed index method. In the experiment, we implement the proposed index on a well-known CDM system, Cassandra, and we compare the performance of the proposed index with the existing index methods. To evaluate the effect of skewed spatial data with different distributions and to study scalability on different index methods, we present a synthetic data generation. We also study the effect of parameters used in the proposed index method in the experiments. The experimental results demonstrate that the proposed index outperforms the existing index methods, especially under the skew data distributions.

We summarize the contributions of this paper as follows:

- We propose an efficient multi-dimensional index structure, the KR$^+$-index, on CDMs to support efficient multi-attribute accesses of skewed data.
- Based on the KR$^+$-index, we define new efficient spatial query algorithms, range queries and k-NN queries.
- The KR$^+$-index uses the characteristics of CDMs effectively.
- The experiments show that the proposed KR$^+$-index outperforms the-state-of-the-art methods.

The remainder of the paper is organized as follows. First, Section 2 presents the preliminaries of multi-attribute access and multi-dimensional index techniques. Section 3 presents a novel multi-dimensional KR$^+$-index. Section 4 presents an analysis of the performance of the proposed KR$^+$-index for multi-attribute accesses on a CDM. Section 5 presents the related studies of CDMs, traditional index techniques, and index techniques on CDMs. Finally, Section 6 concludes the paper.

2. Preliminaries

2.1. Multi-attribute access

For multi-dimensional data search, multi-attribute access is used to restrict multiple attributes at the same time. For instance, *Range Query* and *k-NN Query* are common queries of multi-attribute access and are widely used in location-based services.

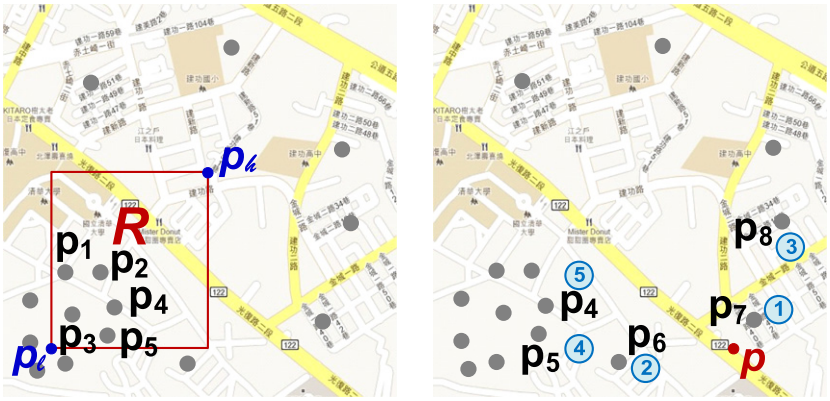
2.1.1. Range queries

Given a set of data points P and a spatial range R, a range query can be formulated as "searching the data points in P that are located in the spatial range R". Note that, in this paper, each data point has location information, e.g., a longitude and a latitude. Without loss of generality, in this paper, a spatial range is represented by a rectangular range. For instance, in Fig. 1(a), given 15 restaurants, marked by gray points, and a red query range R, the range query is to search for which restaurants are located in the range R. As shown in Fig. 1(a), the result of the range query is $\{p_1, p_2, p_3, p_4, p_5\}$.

2.1.2. k-NN queries

Given a set of data points P, a query location $p = (p_x, p_y)$ and a constant k, a k-NN query can be formulated as "searching the data points in P that are the k nearest data points of p". For example, in Fig. 1(b), given 15 restaurants, a user-specified

[1] This article is an extended version of [9].



(a) Range queries. (b) k-NN queries.

Fig. 1. Examples of multi-attribute access.

location p marked by the red color and $k = 5$, a 5-NN query here is to search for five restaurants nearest to p. Thus, the search result of this query is $\{p_4, p_5, p_6, p_7, p_8\}$ as shown in Fig. 1(b).

2.2. Multi-dimensional index techniques

2.2.1. Tree structures

R-trees, developed for indexing multi-dimensional data, are widely used in multi-attribute accesses. Because an R-tree is a balance search tree by dynamically splitting and merging nodes, and can restrict the number of elements in each node by controlling the M and m, it is beneficial for searching skewed data. Moreover, to efficiently index different multi-dimensional data, different variations of R-trees have been developed, such as R^+-tree [10], R^*-tree [11] and the Hilbert R-tree [12]. The R^+-tree developed a new rule of splitting and merging nodes to speed up multi-attribute accesses.

Quad-trees [2] are another common tree structure for indexing multi-dimensional data. In quad-trees, each internal node has exactly four children. However, quad-trees are not balanced trees because a region is split into four sub-regions until the number of data points in the region is less than or equal to a given parameter M.

2.2.2. Linearization

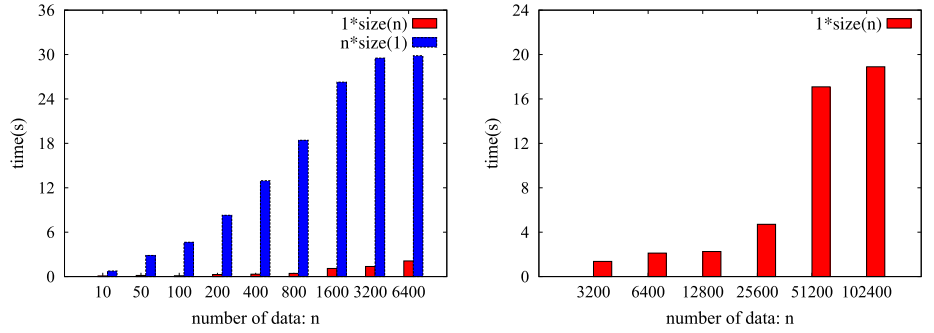
Linearization is a well-known technique for indexing multi-dimensional data by transforming it into one-dimensional data. One of the most popular methods of linearization is using space-filling curves [13], such as a Hilbert curve [14] and Z-ordering [15]. Given two-dimensional data, this method first divides the map into $2^n \cdot 2^n$ non-overlapping grids, where n is a parameter, and assigns a number for each grid according to the order of traversing all grids. Note that the number of each grid is regarded as a key. However, using space-filling curves to index data may not be efficient. If the value of n is set to be lower, it will result in querying more unqualified data, said false-positive, which should be pruned. On the other hand, if n is set to be larger, it would increase the times to retrieve more grids. Thus, for this indexing technique, it is a trade-off to set a proper value of n for efficiency.

3. Multi-dimensional index structure

CDMs provide key-value search, which retrieves a value by a given key, based on the CDM data model. CDMs support basic operations to access data, but these operations do not directly support multi-attribute access. To deal with the problem of multi-attribute access, we have developed a multi-dimensional index structure for CDMs. Furthermore, in this paper, we apply our developed index structure for range queries and k-NN queries on CDMs.

3.1. KR^+-index

Our design of a multi-dimensional index is based on the following observation of CDMs. Fig. 2(a) shows the response time of retrieving a set of n data by two kinds of operations, *Scan* and *Get*. In Fig. 2(a), performing the operation *Scan* once to retrieve n data is more efficient than performing the operation *Get* n times. Fig. 2(b) shows that the response time is increasing dramatically when n is increased from 25,600 to 51,200. We observe three characteristics of CDMs: (1) the time of retrieving n data by one key (i.e., one *Scan*) is less than the time of retrieving n data by n keys (i.e., n *Get*); (2) the time of retrieving n data by one key is large when n is large; and (3) the operation *Scan* is more efficient than multiple *Get* operations when retrieving the same keys. Considering the aforementioned characteristics, for a query we should make the number of



(a) One *Scan* vs. *n Get*. (b) *Scan* with large *n*.

Fig. 2. The features of the CDMs.

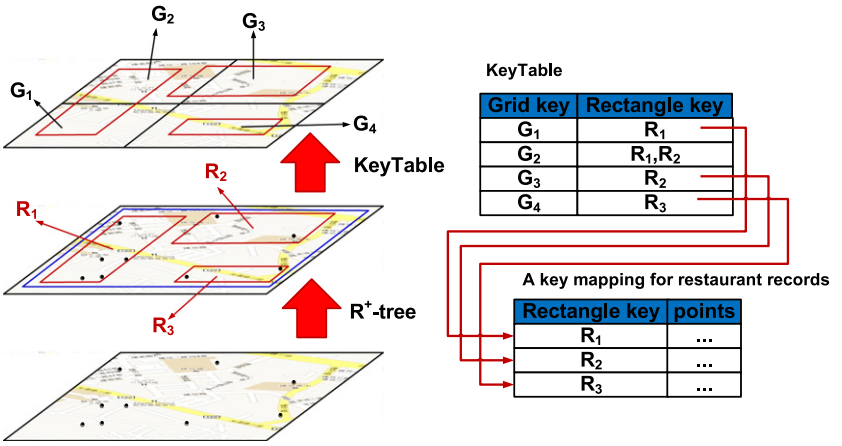


Fig. 3. Overview of the KR$^+$-index.

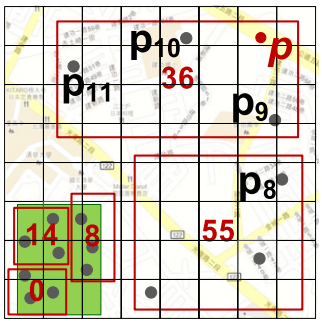
false-positives be smaller from the characteristic 2 and let the number of sub-queries be smaller from the characteristic 1. An R$^+$-tree is a balanced tree that has M and m to control the size of each dynamic rectangle. We could therefore use the M and m to meet the trade-off between false-positive and sub-queries. Considering the characteristic 3, we use the Hilbert curve to let the queried key be as continuous as possible and then the rate of *Scan* is increased.

Fig. 3 is the framework of the KR$^+$-index. First, the data is constructed by the R$^+$-tree with given M, m and the restaurant records for each rectangle, $\{R_1, R_2, R_3\}$, are maintained. In order to retrieve the restaurant records efficiently, we propose a mapping method for retrieving the queried rectangle keys. Second, the map is divided into uniform $2^n \times 2^n$ non-overlapping grids, $\{G_1, G_2, G_3, G_4\}$. Then, for each grid we maintain a list of rectangles that overlap it. For instance, the grid G_2 overlaps rectangles $\{R_1, R_2\}$ so the KeyTable stores a record $\langle G_2, \{R_1, R_2\}\rangle$. Thus, a query could conveniently transform into which grids need to be queried and then through the KeyTable could easily get the required rectangles.

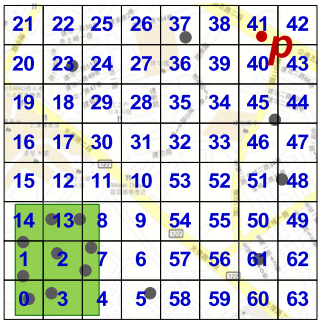
For these key-value storages, it is crucial to define the key, because we use the key to access corresponding data. We construct the R$^+$-tree to discover non-overlapping minimum bounding rectangles. Considering characteristic 3, we use a Hilbert-curve to define the keys because this method manifests superior data clustering compared with other multi-dimensional linearization techniques. For each leaf rectangle, we use the Hilbert-value of the geographic coordinate of the centroid of the rectangle as the key. Note that different rectangles may have the same key, since their central points fall on the same grid. Then, we split the space into uniform non-overlapping $2^n \times 2^n$ grids each of which has a Hilbert-value which is transformed by a Hilbert-curve. Take Fig. 4(a) for example, where each rectangle is given a Hilbert value. Each grid is also given a Hilbert value, and grid 1 in Fig. 4(b) overlaps with the rectangles {0, 14} so that $\langle 1, \{0, 14\}\rangle$ is stored in the KeyTable as shown in Fig. 4(c). We could get the rectangle information though the KeyTable and the multi-attribute access can retrieve the data efficiently. However, the non-leaf nodes are not used in the search, but used in the insertion and deletion as will be illustrated in the next subsection.

3.2. Insertion and deletion

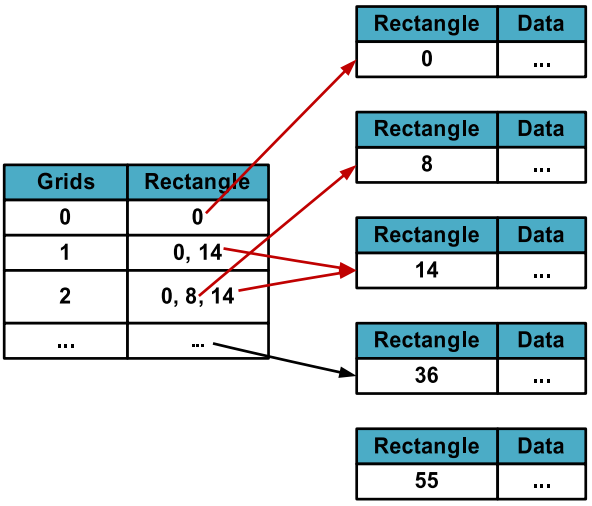
The algorithm to insert a new data point is shown in Algorithm 2. It first loops up, using Algorithm 1, the key of the node corresponding to the node to which the point belongs, and then inserts the data point into the node. Since there is an upper bound to the number of points in the node, the insertion algorithm checks the current size of the node to determine if a



(a) A key definition for rectangles.



(b) A key definition for grids.



(c) A KeyTable for rectangles.

Fig. 4. An example of the KR$^+$-index.

Algorithm 1 Subspace Lookup

```
1: /* p = (x, y) is a data point. */;
2: /* o is the Hilbert order. */;
3: i←x mod o;
4: j←y mod o;
5: return Hilbert(i,j);
```

Algorithm 2 Insert a new data point

```
1: /* p is a new data point. */;
2: key← SubspaceLookup(p);
3: InsertToKRPlust(key,p);
4: if Size(key)>MaxNodeSize then
5:     SplitSpace(key);
6: end if
```

Algorithm 3 Delete a data point

```
1: /* p is a new data point. */;
2: key← SubspaceLookup(p);
3: DeleteFromKRPlust(key,p);
```

split is needed. The deletion algorithm shown in Algorithm 3 is similar to the insertion. It first loops up the key of the node corresponding to the node to which the point belongs, and then deletes the data point from the node.

Algorithm 4 Split node

```
1: /* ns is a node of R+-tree */;
2: /* na, nb are two new nodes split from node ns by R+-tree */;
3: keyOfNa←Hilbert(center points of na);
4: keyOfNb←Hilbert(center points of nb);
5: for each point p in ns do
6:     if p in na then
7:         pointsOfNa.add(p);
8:     else
9:         pointsOfNb.add(p);
10:    end if
11: end for
12: Insert(keyOfNa, pointsOfNa);
13: Insert(keyOfNb, pointsOfNb);
14: Delete(keyOfNs);
```

Algorithm 5 Range Query

Input: p_l, p_h: the range for the query;
Output: points contained in the range;

```
1: Coordinate ← ComputeCoordinateOfGrid(p_l, p_h);
2: Keys ← φ;
3: RectKeys ← φ;
4: Result ← φ;
5: for each Coordinate c ∈Coordinate do
6:     GridKeys ← GridKeys ∪ComputeContainGridKeys(c);
7: end for
8: RectKeys← GetRectKeys(GridKeys);
9: for each Key k ∈RectKeys do
10:    Result ← Result ∪GetContainPoints(k);
11: end for
12: return Result;
```

Algorithm 6 *GetRectKeys*(GridKeys)

Input: GridKeys: the grid keys overlap with query range;
Output: the rectangle keys overlap with query range;

```
1: RectKeys ← φ;
2: for each grid key gk ∈GridKeys do
3:     RectKeys ← RectKeys ∪ KeyTable(gk);
4: end for
5: return RectKeys;
```

3.3. Space split

The R$^+$-tree limits the number of points contained in each node; a node is split when its number of points exceeds this limit. We set the maximum number of points in a node as 50, 100, and 250, and the maximum number of points of insertion performance is set to 50 in the experiments. A split in the R$^+$-tree relies on the key definition of each node, named with the Hilbert value. A node split in the R$^+$-tree will insert two new sub-nodes and delete the old node so that the points in the old node will be allocated into one of the new sub-nodes. The number of new nodes created depends on the index structure used: an R$^+$-tree splits a node in one dimension; the opposite is to let the data points determine the hyperplane, as the *k*-d trees [1] or *k*-d-b trees [16] do. For every dimension split, the name of the new sub-node is created by the Hilbert code of the center points of the new sub-node. Algorithm 4 shows the pseudo code for sub-node name generation following a split.

3.4. Range query

Multi-dimensional range queries are commonly used in location based applications. Algorithm 5 is the pseudo code for a range query in HBase and Cassandra. (p_l, p_h) is the range for the query, p_l is the lower bound and p_h is the upper bound. The Hilbert curve splits the space into grids, and each grid has one grid key. The algorithm first computes the coordinate of grids overlapping the range query. The GridKeys is the set of grid keys contained in the query range. For each coordinate of grid c, the function ComputeContainGridKeys() computes the corresponding grid keys via the Hilbert curve and adds it to the list, GridKeys. Then, according to the key table, we could find the rectangle keys in the query range. Lines 5–8 find the queried key and lines 9–10 fetch the points in the corresponding key. The function GetContainPoint() returns the queried data by first retrieving points from Cassandra and HBase with key k and then filtering out some points that are not in the query range.

Algorithm 7 *k*-NN Query

Input: k: k nearest neighbors; $p = (x, y)$: query point;
Output: k nearest neighbors of (x, y);

```
1: K← φ;
2: QueryRect ← φ;
3: dist ← 0;
4: Rect_scanned ← φ;
5: loop
6:     if QueryRect== φ then
7:         Rect_next ←RectInRegion(p,dist)−Rect_scanned;
8:         for each Rectangle R∈Rect_next do
9:             Push(R, MinDist(p, R), QueryRect);
10:        end for
11:    end if
12:    R ←Pop(QueryRect);
13:    for each Point t ∈R do
14:        K ← K ∪ < t, Dist(p, t) > and sort K by dist;
15:    end for
16:    if dist(k-th point in K, p) ≤ dist then
17:        break;
18:    end if
19:    Rect_scanned ←Rect_scanned ⋃R;
20:    dist←Max(dist, MaxDist(p, R));
21: end loop
22: return K;
```

Algorithm 8 *RectInRegion*(*p*,dist)

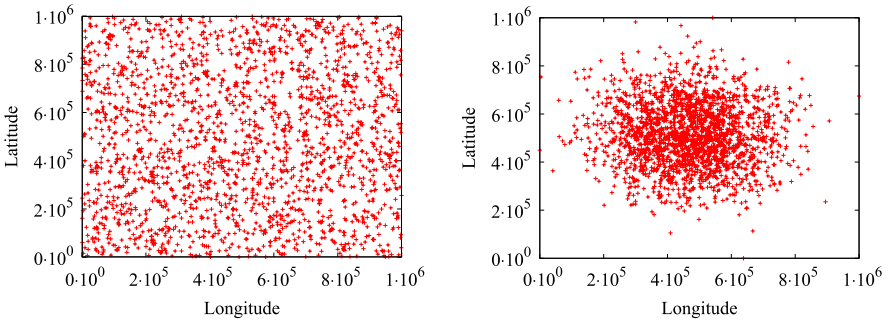
Input: $p = (x, y)$: query point; dist: means a square with edge length 2·dist and with p as its centroid $o = order$: the order of Hilbert;
Output: the keys of rectangles overlap with the input rectangle

```
1: RectKeys ← φ;
2: xl ←(x-dist) mod o;
3: xh ←(x+dist) mod o;
4: yl ←(y-dist) mod o;
5: yh ←(y+dist) mod o;
6: for i=xl → xh do
7:     for j=yl → yh do
8:         GridKeys ← GridKeys ∪ Hilbert(i, j);
9:     end for
10: end for
11: return RectKeys←KeyTable(GridKeys);
```

As shown in Fig. 4, taking the green block as the range query, we will show an example of how range query works. Using the query range to get the geographic coordinates of the overlapped grids, {(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)}, then get the Hilbert values of each geographic coordinate, {0, 3, 4, 1, 2, 7, 14, 13, 8}. Second, get the keys of the rectangles through the KeyTable that grid 0 maps to rectangle {0}, grid 1 maps to rectangles {0, 14}, grid 2 maps to rectangles {0, 8, 14}, etc. Thus, we can get the queried rectangle keys, {0, 8, 14}, by joining the rectangle sets got from the former steps. Finally, use the rectangle keys to retrieve data in the CDMs and then prune the unqualified data to get the query result.

3.5. *k*-NN query

The k-NN query is also commonly used in location based applications. Algorithm 7 shows the k-NN query algorithm in HBase and Cassandra, K stores the result k nearest neighbors, QueryRect stores the rectangles which could be scanned, dist is the range for the rectangle search, $\text{Rect}_{scanned}$ stores the rectangles that had been scanned, and the data structure of QueryRect is a queue. The k-NN query has two main parts: (1) set a range dist to search for rectangles which overlap a square range with centroid p and edge length $2 \cdot$ dist; (2) pick the nearest rectangle of p that is not scanned and add the nearest points in this rectangle into K. The algorithm keeps repeating steps 1 and 2 until the distance of the k-th nearest point and p is less than or equal to dist. Part (1) in Algorithm 7 is in lines 6–11, where RectInRegion() is used to find the rectangles in the square range and line 9 pushes the rectangles that have not been scanned into QueryRect; (2) is in lines 12–18, where line 12 pops the nearest rectangle, and line 14 will add the points of R into K. The function RectInRegion(c, dist) in Algorithm 8 finds the rectangles which overlap with the input square. It is designed by our method for defining the key in the rectangles. Lines 6–8 find the grid keys which overlap the squares, and line 11 returns the rectangles which overlap grids through checking the KeyTable.



(a) Uniform distribution. (b) Normal distribution.

Fig. 5. Distribution of synthetic data.

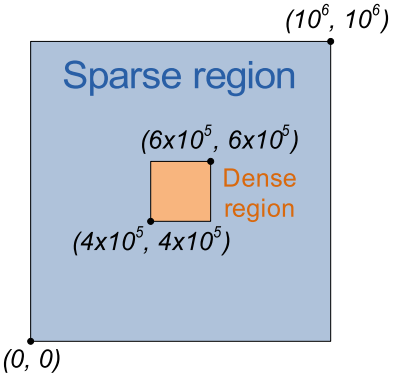


Fig. 6. The sparse and dense regions of synthetic data.

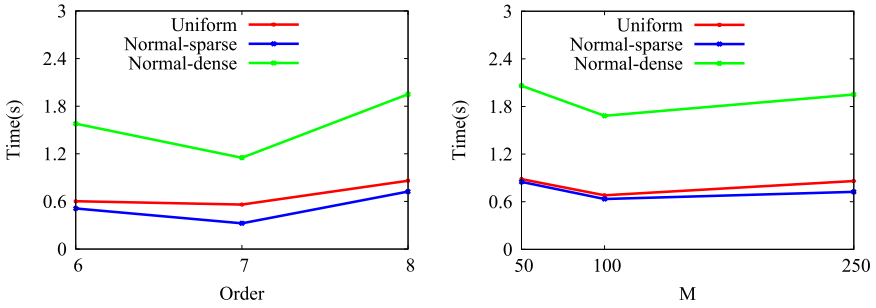
As shown in Fig. 4, take p as the query point, $k = 3$ and given an initial dist $= 0$. First, we will get a rectangle 36 through the KeyTable with a square range of length $2 \cdot$ dist and then insert the location points $\{p_9, p_{10}, p_{11}\}$ of rectangle 36 into K. In that location points are ordered by the distance from p. Second, resize the dist to the minimum distance of k-th/|K|-th location points in K from p, the dist $=$ dist(3rd location point in K, p) in this example. The algorithm continues the first and second steps, it will add the rectangle 55 into Rect_{next} and add the location points in rectangle 55 into K. The algorithm is stopped by dist(3rd location point in K, p) $\leq$ dist, and we get the first three location points $\{p_{10}, p_9, p_8\}$ in K as the query result.

4. Experiment

This section presents the experiments on the response time of range query and k-NN query on Cassandra with the different implementations, a linearization index technique, Hilbert curve (*Hilbert*), and the proposed KR$^+$-index. We also compare the proposed index method with MD-HBase (*MD*) [17,18]. Our experiments were performed on a ring of Cassandra 1.0.10 of ten nodes where each two nodes were on a physical machine. Each physical machine consists of two virtual machines, 2 GB memory and 500 GB HDD and 64 bit Ubuntu 8.04.4.

Our evaluation uses synthetically generated data sets primarily due to the need for huge data sets (gigabytes) and the need to control data distribution to study the effect of skewness on the performance of the proposed index. The synthetic data generator proposed by Yaling and Osmar [19] has two kinds of distribution, *normal* and *uniform*. The multivariate uniform distribution data is simply generated with each one-dimensional uniform distribution separately, since the joint distribution of two or more independent one-dimensional uniform distributions is also uniform. The multivariate normal distribution data is generated by first producing two-dimensional uniform distribution then using the Box–Muller transformation [20,21] to transform the two-dimensional uniform distribution to a two-dimensional bivariate normal distribution with mean $\mu = 0$ and variance $\sigma^2 = 1$. We generated synthetic data of uniform and normal distribution of one cluster with data size equal to two hundred thousand, five hundred thousand and one million on a square map of one million units of length, as shown in Fig. 5(a) and (b).

We generated six datasets using a model of normal and uniform distribution that for each distribution generates sizes of two hundred thousand, five hundred thousand and one million points. The data generated in uniform distribution is denoted as *Uniform*. To study the effect of skewness, as shown in Fig. 6, we discriminate query scopes according to data generated in normal distribution as follows. The data located in the dense region on the map center with ten hundred thousand units



(a) The response times for the KR$^+$-index of varying order.

(b) The response times for the KR$^+$-index of varying (M, m).

Fig. 7. The effect of parameters in the KR$^+$-index.

Table 1
The relationships between the range query response time and the parameters of the KR$^+$-index.

(a) Average length and width of rectangles of KR$^+$-index				(b) The length and width of grids with different order	
M	m	Avg. len.	Avg. wid.	Order	Len.
50	25	1128.4283	2656.2368	6	15 625
100	50	4928.2417	4671.1948	7	7812.5
250	125	6941.6216	6280.1025	8	3906.25

of length is denoted as *Normal-dense*. The data located in the sparse region outside the dense region is denoted as *Normal-sparse*. In the experiments, for each data, we performed one hundred random queries and averaged response times.

4.1. Parameter study of KR$^+$-index

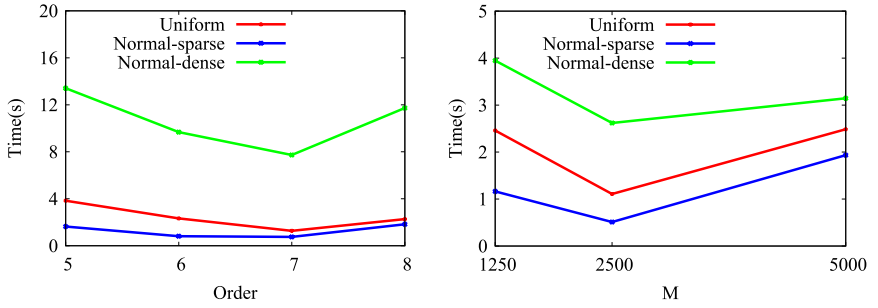
This section presents a way of setting parameters in the proposed KR$^+$-index. The KR$^+$-index method has three parameters, the lower bound and upper bound of rectangles (M, m), and the order o. Fig. 7 shows the effect of these parameters on the response time of the range query. The lower bound and upper bound of rectangles we evaluated in the range query of the KR$^+$-index are $(M, m) = \{(50, 25), (100, 50), (250, 125)\}$, and the order are $o = \{6, 7, 8\}$. Fig. 7 plots the range query response times for the KR$^+$-index of varying query size, order, data size and (M, m). Fig. 7(a) plots the range query response times of varying order with fixed data size $ds = 1\,000\,000$, query square size of length $qs = 100\,000$ and $(M, m) = (250, 125)$. Fig. 7(b) plots the range query response times of varying (M, m) with fixed query square size of length $qs = 100\,000$, data size $ds = 1\,000\,000$ and order $o = 8$. As shown in Fig. 7(a), there is a trade-off between the false-positive ratio and the number of sub-queries as varying order. The response time increased when the order was larger than 7, owing to spending a lot of time in fetching sub-queries; the response time increased when the order was less than 7 due to spending much time pruning points not in the query range. It is the same as the range query response time of varying (M, m).

The decision of (M, m) and order affect the efficiency of the spatial query. We then determine the proper values of (M, m) and order automatically in the following way. We knew that (M, m) influences the size of the rectangles and the order decides the grid size. We observed the relationship between the response times and the parameters, (M, m) and o. As shown in Table 1, this is the average length and width of rectangles of ten million data size and the length of grids with different order; the average length and width of $(M, m) = (250, 125)$ is close to the grid length of order 7, and the range query response times of the KR$^+$-index, shown in Fig. 7(a), expressed that the range query with fixed $(M, m) = (250, 125)$ had better response time as order $o = 7$. We found that the closer the rectangle size and the grid size, the better the response time of the range query. Thus, we first decide a small value of (M, m), and evaluate the average size of the rectangles. We then calculate the closest grid size generated by order. The objective function of o can be expressed as $o = \min_o(|len(o) - avgLen(M)| + |len(o) - avgWid(M)|)$. About the setting of (M, m), we evaluate the range query of varying $(M, m) = \{(50, 25), (100, 50), (250, 125), (1250, 625), (2500, 1250), (2500, 5000)\}$. The (M, m) has a lower bound at $(100, 50)$, the response times of range query was considerably worse as $(M, m) = \{(1250, 625), (2500, 1250), (2500, 5000)\}$, but the response time had a slight increase as $(M, m) = (50, 25)$.

4.2. Performance comparison

4.2.1. Range query

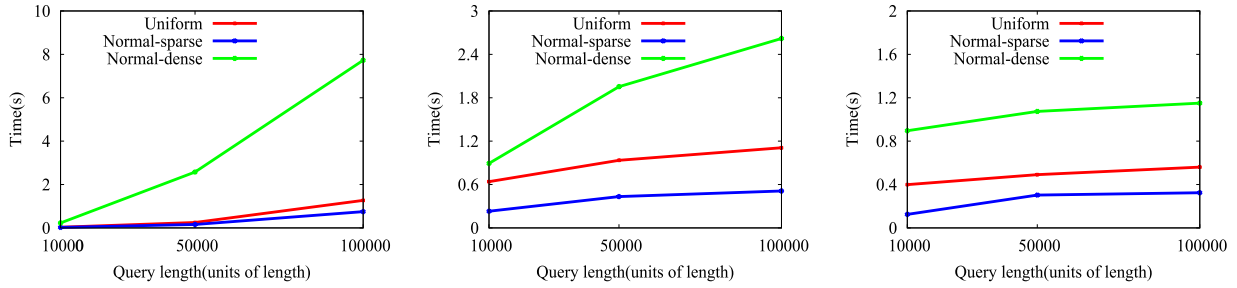
The section presents a performance comparison through range query. We compare the proposed index method (KR) with two existing index methods, Hilbert and MD. We evaluate the range query with a square size equals to ten thousand units of length, five thousand units of length and one million units of length. Before performance comparison, we first study the



(a) The range query response time for Hilbert of varying order.

(b) The range query response time for MD of varying M.

Fig. 8. The parameter determination for the index methods, Hilbert and MD.



(a) Hilbert. (b) MD. (c) KR$^+$-index.

Fig. 9. The response time of the Hilbert, MD, and KR$^+$-index with varying query size.

effect of parameters of existing index methods, Hilbert and MD, on performance, and set proper values of the parameter of these methods. We then compare these index methods with various query sizes qs and data sizes ds.

Parameter determination for Hilbert and MD. The Hilbert method has a parameter order o, which is used to decide how many grids were divided from the map. The order we evaluated in the range query for Hilbert is $o = \{5, 6, 7, 8\}$. Fig. 8(a) plots the range query response times of varying order with fixed query square size of length $qs = 100\,000$ and data size $ds = 1\,000\,000$. In light of Fig. 8(a), there is a lower bound when $o = 7$, the response time is increased when the order exceeds 7 and when it is lower than 7. This is reasoned by the trade-off between the false-positive ratio and the number of sub-queries. This means that the number of sub-queries increased as the order exceeded 7 and the false-positive ratio increased as the order fell below 7; thus both led to an increase in the response time. We then set $o = 7$ for the Hilbert index method in the following experiments.

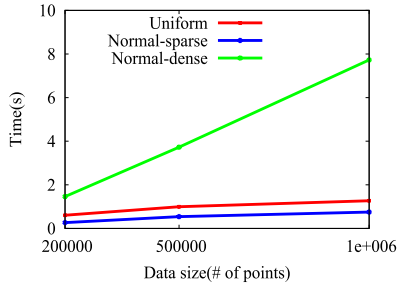
The MD, proposed by Shoji et al., has a parameter M, used to decide the upper bound of grids. The upper bound M we evaluated in the range query of the MD is $M = \{1250, 2500, 5000\}$. Fig. 8(b) plots the range query response times of varying M with fixed data size $ds = 1\,000\,000$ and query square size of length $qs = 100\,000$. In the light of Fig. 8(b), we then set $M = 2500$ for the MD index method in the following experiments.

Effect of query size. Fig. 9(a) plots the range query response times of varying query size with fixed data size $ds = 1\,000\,000$ and order $o = 7$. As shown in Fig. 9(a), the larger the query size, the larger the response time, since the number of fetched points increases. Fig. 9(b) plots the range query response times of varying query size with fixed data size $ds = 1\,000\,000$ and $M = 2500$. In Fig. 9(b), the response time increased as the query size increased because of the fetched points increasing. Fig. 9(c) plots the range query response times of varying query size with fixed data size $ds = 1\,000\,000$. As shown in Fig. 9, KR$^+$-index outperforms the two index methods.

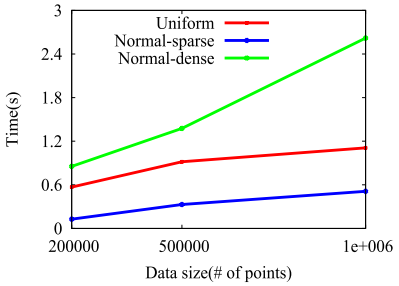
Effect of data size. Fig. 10(a) plots the range query response times of varying data size with fixed query square size of length $qs = 100\,000$ and order $o = 7$. In Fig. 10(a), the response time increased as the data size increased. Fig. 10(b) plots the range query response times of varying data size with fixed query square size of length $qs = 100\,000$ and $M = 2500$. In Fig. 10(b), the response time increased as the data size increased because of the fetched points increasing. Fig. 10(c) plots the range query response times of varying data size with fixed query square size of length $qs = 100\,000$. In Fig. 10, KR$^+$-index outperforms the other two index methods. The response times of Hilbert and MD are worse than the KR$^+$-index when the data are skewed (i.e., Normal-dense).

4.2.2. *k*-NN query

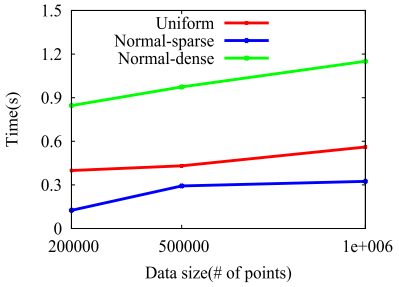
This section presents a performance comparison through k-NN query. We compare the proposed index method (KR) with two existing index methods, Hilbert and MD. We evaluate k-NN query with a square size equals to ten thousand units of



(a) Hilbert.

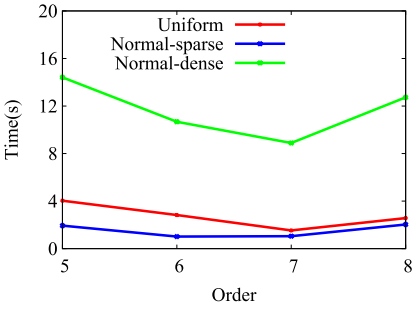


(b) MD.

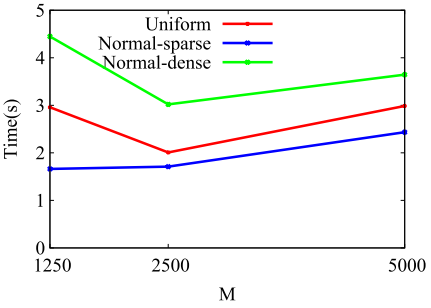


(c) KR$^+$-index.

Fig. 10. The response time of the Hilbert, MD, and KR$^+$-index with varying data size.



(a) The k-NN query response times for Hilbert with varying order.



(b) The k-NN query response times for MD with varying M.

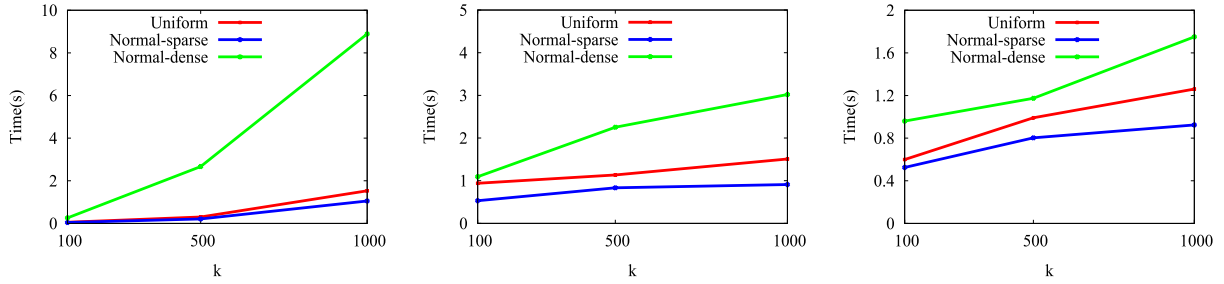
Fig. 11. The parameter determination for the index methods, Hilbert and MD.

length, five thousand units of length and one million units of length. The parameter k is set as 100, 500, and 1000. Before performance comparison, we first study the effect of parameters of existing index methods, Hilbert and MD, on performance, and set proper values of the parameter of these methods. We then compare these index methods with various query sizes qs and parameter k.

Parameter determination for Hilbert and MD. Similarly, the orders we evaluated in k-NN query of Hilbert are $o = \{5, 6, 7, 8\}$. Fig. 11(a) plots the range query response times of varying order with fixed $k = 1000$ and data size $ds = 1\,000\,000$. In the light of Fig. 11(a), we set $o = 7$ for the Hilbert index method in the following experiments. The parameter M of MD was set as $M = \{1250, 2500, 5000\}$. Fig. 11(b) plots the range query response times of varying M with fixed data size $ds = 1\,000\,000$ and $k = 1000$. In the light of Fig. 11(b), we set $M = 2500$ for the MD index method in the following experiments.

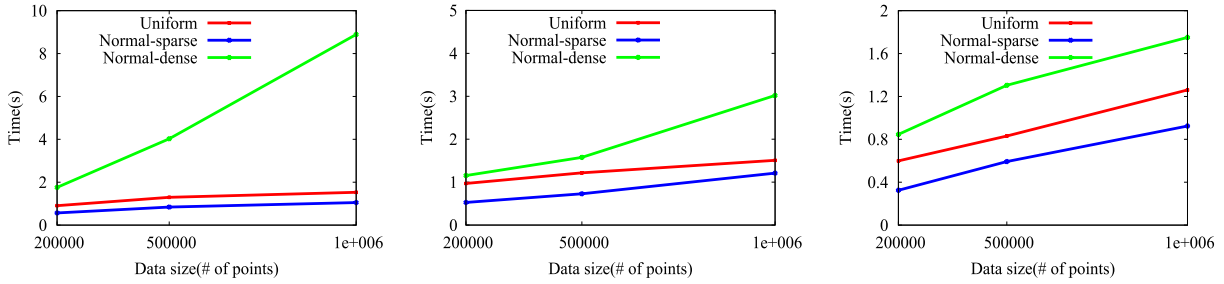
Effect of k. Fig. 12(a) plots k-NN query response times of varying query size with fixed data size $ds = 1\,000\,000$ and order $o = 7$. As shown in Fig. 12(a), the larger k, the larger the response time, since the number of fetched points increasing. Fig. 12(b) plots k-NN query response times of varying query size with fixed data size $ds = 1\,000\,000$ and $M = 2500$. In Fig. 12(b), the response time increased as k increased because of the fetched points increased. Fig. 12(c) plots k-NN query response times of varying k with fixed data size $ds = 1\,000\,000$. As shown in Fig. 12, KR$^+$-index outperforms the other two index methods.

Effect of data size. Fig. 13(a) plots k-NN query response times of varying data size with fixed $k = 1000$ and order $o = 7$. In Fig. 13(a), the response time increased as the data size increased. Fig. 13(b) plots k-NN query response times of varying data



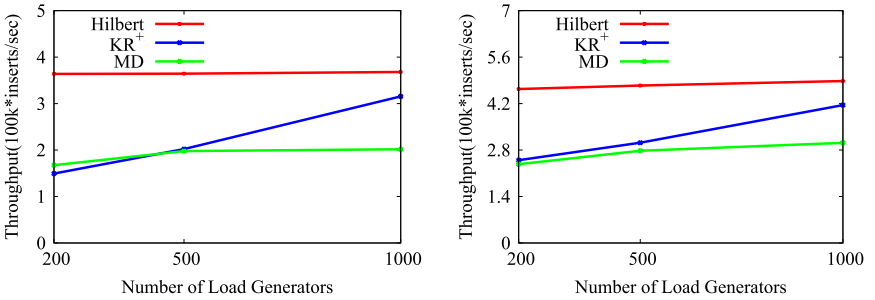
(a) Hilbert. (b) MD. (c) KR$^+$-index.

Fig. 12. The response time of the Hilbert, MD, and KR$^+$-index with varying query size.



(a) Hilbert. (b) MD. (c) KR$^+$-index.

Fig. 13. The response time of the Hilbert, MD, and KR$^+$-index with varying query size.



(a) The insert throughput as a function of the load on the system. (b) The delete throughput as a function of the load on the system.

Fig. 14. The insertion and deletion throughput.

size with fixed $k = 1000$ and $M = 2500$. In Fig. 13(b), the response time increased as the data size increased because of the fetched points increasing. Fig. 13(c) plots k-NN query response times of varying data size with fixed $k = 1000$. In Fig. 13, KR$^+$-index still outperforms the two index methods.

4.3. Performance study of insertion and deletion

Supporting high insert throughput of data updates is critical for sustaining the large numbers of location based services. We evaluated the insert performance on a Cassandra cluster with ten commodity nodes. Fig. 14(a) plots the insert throughput as a function of the load on the system. We varied the number of load generators as 200, 500, and 1000; each generator created a load of 1000 inserts per second. We use the synthetic data using a Normal distribution with mean $\mu = 0$ and variance $\sigma^2 = 1$. Using synthetic data allows us to control the skew of large data sets. All of the methods shown good scalability; the throughput is at least 150k location data points per second. The lower throughput of the KR$^+$-index and MD is the cost associated with the splitting nodes on the R$^+$-tree and the quad tree. On the other hand, the Hilbert does not need splitting nodes. On average, the KR$^+$-index needs about 25 s to split a node and the MD needs about 40 s. The dataset and the number of load generators used in deletion is the same as for insertion. The delete throughput, as shown in Fig. 14(b) is higher than the insert throughput, but the performance comparison seems similar for Hilbert, KR$^+$-index and MD. The KR$^+$-index and MD have lower throughput since there is a greater cost of merging nodes on the R$^+$-tree and the quad tree.

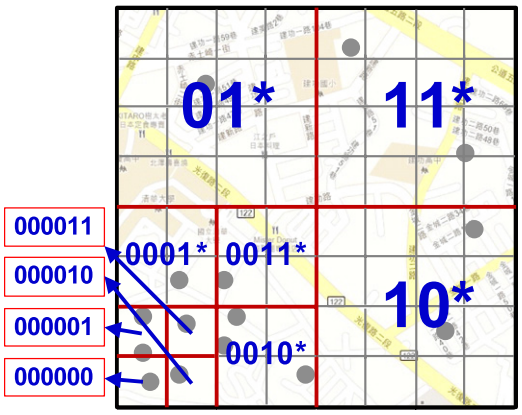


Fig. 15. A key formulation in MD-HBase.

5. Related work

To support efficient multi-attribute accesses, R-tree was developed and is widely used for indexing multi-dimensional data. To accelerate the performance of multi-dimensional data, Kamel et al. [22] proposed parallel R-trees, a hardware architecture consisting of one processor with several disks attached to it. The study focuses on maximizing parallelism for multiple range queries of concurrent users on one workstation. Moreover, Wang et al. [23] presented a parallel R-tree search algorithm running on distributed shared virtual memory for parallel I/O and CPU operations on workstations. In addition, Lai et al. [24] proposed an upgraded parallel R-tree model to balance the load among processors and to decrease the conflicts of intra- and inter-transactions in the environment of workstations. These studies focus on processing a R-tree in a parallel way over multiple disks. However, in this paper, we focus on developing a multi-dimensional index structure and designing the key names for supporting efficient key-value store accesses in CDMs and supporting the functionality needed in location-based services.

Nowadays, CDMs provide high scalability for a large data and location based services. Most NoSQL CDMs are developed based on a BigTable management system, and the MapReduce framework [25], such as Hadoop [26], is commonly used for the BigTable-like management system in the cloud. Moreover, SpatialHadoop [27], an open source MapReduce framework, was introduced recently and designed specifically to deal with large spatial data. SpatialHadoop can support spatial data types and spatial indexes, including grid file and R-tree. Although the MapReduce framework can provide a parallel processing, the inherent restriction of a BigTable data structure induces that the developed CDMs only support some basic operations, e.g., *Get* and *Scan*, but cannot support efficient multi-attribute accesses for the functionality needed in location-based services without well-designed key names.

Recently, there has been an increasing amount of work on constructing indexes on CDMs. B-tree is a commonly used index structure. Wu et al. [28] presented a scalable B-tree based indexing scheme which builds a local B-tree for the dataset stored in each compute node and builds a Cloud Global index, called the CG-index, to index each compute node. However, the B-tree index cannot support multi-dimensional queries effectively. Besides, much work on the R-tree index structure for multi-dimensional data has been done, such as [29–31]. Wang et al. [29] presented RT-CAN, a multi-dimensional indexing scheme. RT-CAN is built on top of local R-tree indexes and dynamically selects a portion of the local R-tree nodes to publish on the global index. Although it used R-tree indexing, it built the R-tree on their own distributed system epiC. Zhang et al. [30] combined R-tree and k-d tree to be the index structure, and Liao et al. [31] presented an approach to construct a block-based hierarchical R-tree index structure. These works all build an index structure on the Hadoop distributed file system or on Google's file system to support multi-dimensional queries.

MD-HBase [17,18] is a data management system, based on HBase, using Quad trees and k-d trees coupled with Z-ordering to index multi-dimensional data for LBSs. The keys of MD-HBase are the Z-values of the dimensions being indexed. It uses the trie-based approach for splitting equal-sized space and builds Quad tree and k-d tree index structures on the key-value data model. Moreover, MD-HBase proposed a novel naming scheme, called longest common prefix naming, for efficient index maintenance and query processing. Although the experiment of MD-HBase shows that the proposed indexing method is efficient for multi-dimensional data, it has some constraints. Before describing these constraints, we have discovered a characteristic of cloud managements for data accesses through experiment. A trade-off exists between the number of points for getting one key and the number of keys for scanning; a reduction in the number of points for getting one key results in an increase in the number of keys for scanning and vice versa. The way of splitting the space of the Quad tree and k-d tree is fixed, which may make some nodes store zero point. In addition, the Quad tree and the k-d tree cannot balance the number of stored points for each node, because they do not restrict the minimum number of points in one space. Therefore, if we regard one node as one key, it will make the keys store unbalanced data points, especially as the data is not uniform. Fig. 15 is a Quad tree example of space splitting for MD-HBase. According to the data points in the map, the Quad tree will

split the whole space into three. The red line shows the splitting results, and each black grid has its Z-ordering value. For instance, the Z-ordering value of (0, 0) is 000000 and (1, 0) is 000010. Then, the key of each region split by read line is the prefix of the Z-ordering value of its sub-regions. Consequently, there are 10 keys, 000000, 000001, 000010, 000011, 0001*, 0010*, 0011*, 01*, 10* and 11*. However, there may be no data points in some regions. As mentioned above, the Quad tree and k-d tree cannot deal with multiform distribution data efficiently.

6. Conclusion

We proposed a scalable multi-dimensional index, KR$^+$-index, based on an existing CDM, Cassandra. It supports efficient multi-dimensional range queries and nearest neighbor queries. We used R$^+$ to construct the index structure and designed the key for efficient accessing of data. In addition, we redefined the spatial query algorithm, including range query and k-NN query for the proposed KR$^+$-index. The proposed KR$^+$-index took the characteristics of CDMs into account so that the proposed KR$^+$-index is much more efficient than other index methods in the experiments. The experiments show that the proposed KR$^+$-index outperforms the-state-of-the-art index method, MD-HBase, especially for skewed data.

References

[1] J.L. Bentley, Multidimensional binary search trees used for associative searching, Communications of the ACM 18 (9) (1975) 509–517.
[2] R.A. Finkel, J.L. Bentley, Quad trees a data structure for retrieval on composite keys, Acta Informatica 4 (1) (1974) 1–9.
[3] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, 1984, pp. 47–57.
[4] M. Stonebraker, SQL databases vs. NoSQL databases, Communications of the ACM 53 (4) (2010) 10–11.
[5] A. Khetrapal, V. Ganesh, Hbase and hypertable for large scale distributed storage systems, Dept. of Computer Science, Purdue University, 2008.
[6] A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system, ACM SIGOPS Operating Systems Review 44 (2) (2010) 35–40.
[7] J. Varia, Cloud architectures, Technical Report, Amazon Webservices, 2008.
[8] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: a distributed storage system for structured data, ACM Transactions on Computer Systems 26 (2) (2008) 1–26.
[9] Y.-T. Hsu, Y.-C. Pan, L.-Y. Wei, W.-C. Peng, W.-C. Lee, Key formulation schemes for spatial index in cloud data managements, in: Proceedings of the 13th IEEE International Conference on Mobile Data Management, 2012, pp. 21–26.
[10] T. Sellis, N. Roussopoulos, C. Faloutsos, The R$^+$-tree: a dynamic index for multi-dimensional objects, in: Proceedings of the 13th International Conference on Very Large Data Bases, 1987, pp. 507–518.
[11] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, 1990, pp. 322–331.
[12] I. Kamel, C. Faloutsos, Hilbert R-tree: an improved R-tree using fractals, in: Proceedings of the 20th International Conference on Very Large Data Bases, 1994, pp. 500–509.
[13] T. Bially, Space-filling curves: their generation and their application to bandwidth reduction, IEEE Transactions on Information Theory 15 (6) (1969) 658–664.
[14] A.R. Butz, Convergence with Hilbert's space filling curve, Journal of Computer and System Sciences 3 (2) (1969) 128–146.
[15] G.M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, Technical Report, IBM, Ottawa, Canada, 1966.
[16] J.T. Robinson, The kdb-tree: a search structure for large multidimensional dynamic indexes, in: Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, 1981, pp. 10–18.
[17] S. Nishimura, S. Das, D. Agrawal, A. El Abbadi, $\mathcal{MD}$-HBase: a scalable multi-dimensional data infrastructure for location aware services, in: Proceedings of the 12th IEEE International Conference on Mobile Data Management, 2011, pp. 7–16.
[18] S. Nishimura, S. Das, D. Agrawal, A. El Abbadi, $\mathcal{MD}$-HBase: design and implementation of an elastic data infrastructure for cloud-scale location services, Distributed and Parallel Databases 31 (2) (2013) 289–319.
[19] Y. Pei, O. Zaïane, A synthetic data generator for clustering and outlier analysis, Technical Report, TR06-15, Department of Computing Science, University of Alberta, 2006.
[20] G.E.P. Box, M.E. Muller, A note on the generation of random normal deviates, The Annals of Mathematical Statistics 29 (2) (1958) 610–611.
[21] https://en.wikipedia.org/wiki/Box–Muller_transform.
[22] I. Kamel, C. Faloutsos, Parallel R-trees, in: Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, 1992, pp. 195–204.
[23] B. Wang, H. Horinokuchi, K. Kaneko, A. Makinouchi, Parallel R-tree search algorithm on dsvm, in: Proceedings of the Sixth International Conference on Database Systems for Advanced Applications, 1999, pp. 237–245.
[24] S. Lai, F. Zhu, Y. Sun, A design of parallel R-tree on cluster of workstations, in: Proceedings of the International Workshop on Databases in Networked Information Systems, 2000, pp. 119–133.
[25] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.
[26] http://hadoop.apache.org/.
[27] http://spatialhadoop.cs.umn.edu/.
[28] S. Wu, D. Jiang, B.C. Ooi, K.L. Wu, Efficient B-tree based indexing for cloud data processing, Proceedings of the VLDB Endowment 3 (1–2) (2010) 1207–1218.
[29] J. Wang, S. Wu, H. Gao, J. Li, B.C. Ooi, Indexing multi-dimensional data in a cloud system, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 2010, pp. 591–602.
[30] X. Zhang, J. Ai, Z. Wang, J. Lu, X. Meng, An efficient multi-dimensional index for cloud data management, in: Proceeding of the First International Workshop on Cloud Data Management, 2009, pp. 17–24.
[31] H. Liao, J. Han, J. Fang, Multi-dimensional index on hadoop distributed file system, in: Proceedings of the Fifth IEEE International Conference on Networking, Architecture and Storage, 2010, pp. 240–249.