Journal of Database Management
Volume 34 • Issue 1

A Distributed Spatial Index With High Update Efficiency for Location-Based Real-Time Services

Junhua Fang, Soochow University, China
Zonglei Zhang, Soochow University, China*

ABSTRACT

LBS-RT (location-based service in a real-time manner) has become popular because it can provide quick and timely services. Range query is often used in LBS-RT, which finds objects in a specified area, and spatial indices are often used to speed up range query. However, in LBS-RT, there are some difficulties. Spatial index was originally designed to index static dataset, but the dataset is dynamic in LBS-RT, which needs lots of insert and delete operations. To meet the gap, this paper proposes a new distributed spatial index called GQ-QBS. It's a two-layer master-slave mode that consists of a global index and multiple local indices. The global index (GQ-tree) is responsible for the dynamic load balancing and auto-scaling, while the local index (QBS-tree) is for quickly updating and querying. Experiments show the index has a significant advantage in LBS-RT.

KEYWORDS

Auto Scaling, Distributed Computing, Dynamic Load Balancing, Index Technology, Parallel Processing, Real-Time Processing, Spatio-Temporal Data, Update Efficiency

INTRODUCTION

With the rapid development of wireless technology and the ubiquity of portable devices, Location-Based Service (LBS) (Lu et al., 2011; Usman et al., 2018; Zhu et al., 2017) is playing an increasingly important role in our daily life, such as navigation system (Win et al., 2011; Kawamata & Oku, 2019), location-based recommendation (Park et al., 2007; Ye et al., 2010) and traffic congestion prediction (Xu et al., 2020). As the primary data source for LBS, spatial-temporal data is known for its vital timeliness, which means its value decays quickly over time. Therefore, providing LBS-RT (Location Based Service in a real-time manner) by processing spatial-temporal data in real-time has become a hot topic.

DOI: 10.4018/JDM.318454 *Corresponding Author

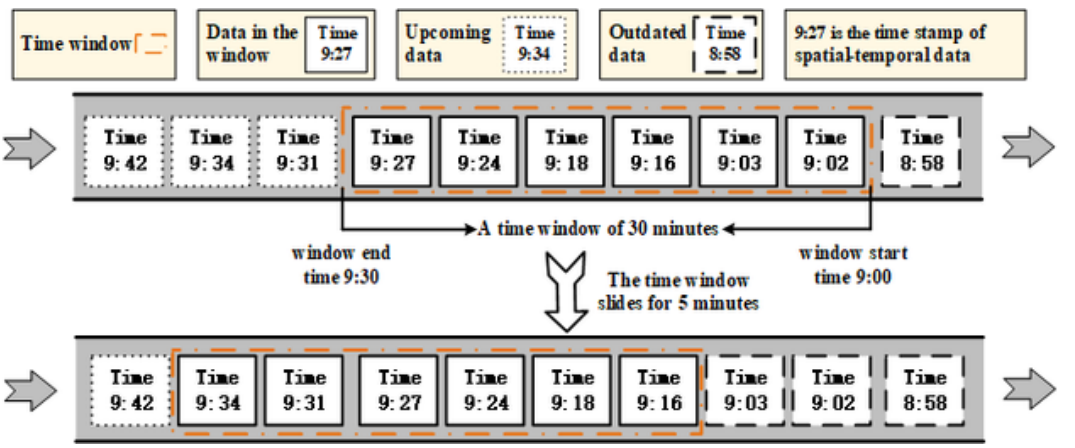
Range query is a basic operation in LBS, and its function is to find all objects in a specified area. Spatial indexes are often used to speed up range query. Similarly, spatial indexes are often used in LBS-RT, and the following example of LBS-RT requires a spatial index. During a parade, the government monitors the size of the crowd in real-time to match the level of security measures. The crowd size can be accurately calculated by clustering the trajectory data generated in the last N minutes. The system needs to update the crowd size regularly to ensure timeliness. Trajectory clustering requires a large number of trajectory similarity searches. Many frameworks (Xie et al., 2017; Shang et al., 2018) that perform similarity search in a large trajectory set require a spatial index to speed up similarity search. LBS-RT only analyzes the data generated in the last N minutes. In Figure 1, those applications often use a sliding time window (Chen et al., 2019) to maintain data. Every time the window slides, outdated items slide out the time window, such as the two items at 9:02 and 9:03, new items slide into it, such as the two items at 9:31 and 9:34. Therefore, the index in LBS-RT needs to perform a lot of update operations.

R-tree (Guttman et al., 1984) and its variants (Leutenegger et al., 1997; A. Fu et al., 2000; Zhou et al., 2008; Kamel et al., 1994; Ciaccia et al., 1997; Y. Fu et al., 2003; Beckmann et al., 1990; Zaschke et al., 2014; Jung et al., 2014; Phan et al., 2017; Amaral et al., 2016; Xiong & Aref, 2006) are commonly used spatial indexes. Most variants are designed to improve the retrieval performance of R-tree, such as STR-tree (Leutenegger et al., 1997), VP-tree (A. Fu et al., 2000), KD-tree (Zhou et al., 2008), Hilbert-tree (Kamel et al., 1994), M-tree (Ciaccia et al., 1997), QR-tree (Y. Fu et al., 2003), R-star-tree (Beckmann et al., 1990) and PH-tree (Zaschke et al., 2014). These indexes don't pay attention to improving update performance. Although some works (Zhang. F et al., 2014; Jo & Jung, 2018) use distributed methods to maintain index for improving update performance, they don't reduce the resource consumption.

To improve the updated performance, the author designed a single-node spatial index called QBS-tree. QBS-tree has good update and query performance. To index large a dynamic spatial data set, the author expands QBS-tree into a distributed spatial index called GQ-QBS. GQ-QBS is a two-layer master-slave distributed spatial index, consisting of a global index (GQ-tree) and multiple local indexes (QBS-tree). QBS-tree is responsible for efficient update and query, while GQ-tree is for the dynamic load balancing and auto-scaling. Experiments compare different spatial indexes on a single node and a distributed cluster, and the results prove that both QBS-tree and GQ-QBS have obvious performance advantages. The following are the main contributions of this paper.

1. The anther proposes a new single-node spatial index, called QBS-tree, with high update efficiency;

Figure 1.
Sliding time window



2. The anther designs a lightweight distributed index to support rapid update and query, and this index has dynamic load balancing and auto-scaling functions;
3. The anther uses massive actual data to test the performance of different spatial indexes.

PRELIMINARIES

Problem Formulation

In LBS-RT, there are two types of input items: the query item and the index item for an index. LBS-RT process input items in the form of a data stream (Akidau et al., 2015), and items in the data stream are time-ordered.In Figure 1, LBS-RT uses a sliding window to maintain a dynamic spatial data set. The problem to be solved in this paper is to find all the index items of the dynamic spatial data set in each query rectangle. This kind of query is called Real-time Range Query. Some notations are used in the rest of this paper, in Table 1.

Solution Overview

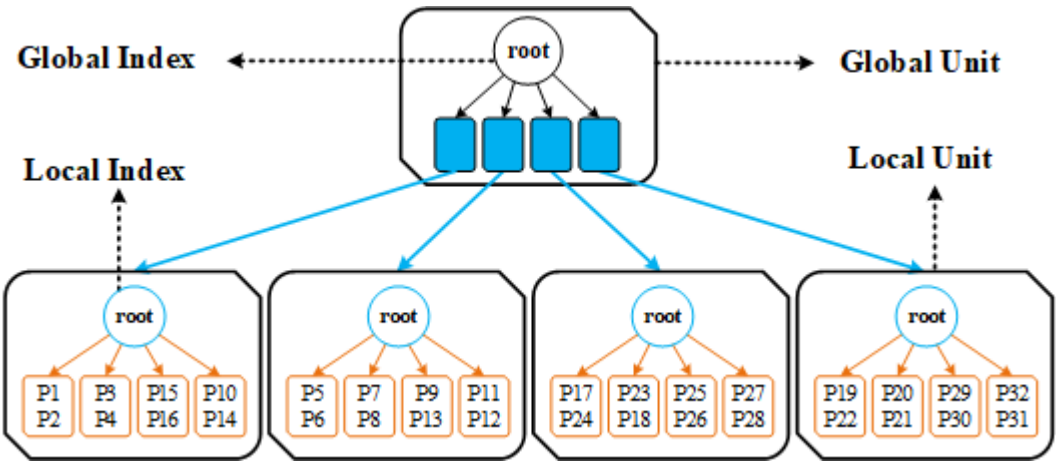
The author solves Real-time Range Query in two steps. Firstly, the author designed a single-node spatial index QBS-tree with high update performance; secondly, the author intended a distributed spatial index GQ-QBS to overcome the single-node performance bottleneck when indexing a large dataset.

Most single-node spatial indexes' update manner are bottom-up to ensure all leaf nodes are on the same layer. It benefits improving query performance. But when there are many update operations in LBS-RT, it makes index frequently adjust its tree structure, which is an expensive operation. On the contrary, QBS-tree's update manner is top-down, which makes the balance factor configurable. The bigger the balance factor, the more QBS-tree can tolerate changes in the spatial distribution of index items, and the lower the frequency of QBS-tree adjusting their tree structure. However, if the balance factor is too big, the query capability will be reduced because the query path (Milo & Suciu, 1999) will become longer. Therefore, the balance factor is designed as a user-configurable parameter to adapt to different application scenarios.

Table 1.
The notations

Notation	Definition
d	The dimension of an index item.
r	The root node of a tree.
l	The leaf node of a tree.
n	The leaf or non-leaf node of a tree.
n[*i*]	When *n* is a non-leaf node, *n*[*i*] is the *i*th child node of *n*.
n.pa	The parent node of *n*.
n.MBR	The MBR of all index items in subtree *n*.
M, m, Mid	*M* and *m* are the upper and lower bound of the number of index items in a leaf node. *Mid* is $\left(m + M\right)\Big/2$.
S	***S*** is a set of index items. When ***S*** have an alphabet subscript it represents the index items stored in some one subtree, such as $\boldsymbol{S}_l$ is the set of index items in the leaf node *l*.
S.*size*	The size of ***S***.

Figure 2.
A distributed spatial index established on the data from Figure 3-(a)



Section GLOBAL INDEX introduces GQ-QBS. In Figure 2, GQ-QBS is a master-slave model which consists of a global index and multiple local indexes. The computation unit that maintains the Global Index (GQ-tree) is called Global Unit(GU), while that maintains the Local Index (QBS-tree) is called Local Unit(LU). The global index recursively divides the global region into several sub-regions, and each leaf node indexes a sub-region. Each local index only indexes the index items in one sub-region. In this way, the update and query can be processed in parallel between LUs. The index items are continuously changing so that data skew may occur between local indexes over time. To achieve load balancing, GQ-QBS periodically adjust GQ-tree.

LOCAL INDEX

Features of QBS-tree

This subsection introduces the following three features of QBS-tree.

1. Each non-leaf node has four children, and all index items are stored in leaf nodes.
2. The number of index items in each leaf node has a lower bound m and an upper bound M. M is five times of m.
3. QBS-tree has a configurable balance factor.

QBS-tree is A Quad-tree

QBS-tree recursively divides a global region into four sub-regions from top to bottom, in Figure 3. The number of index items in each sub-region is approximately the same by the splitting algorithm introduced in Algorithm 1. In Figure 3, a node divides its region into two sub-regions on the y-axis and divides each sub-region into two on the x-axis. The division method mentioned above references Sort-Tile-Recursive, which has proven to be an efficient packing algorithm (Leutenegger et al., 1997).

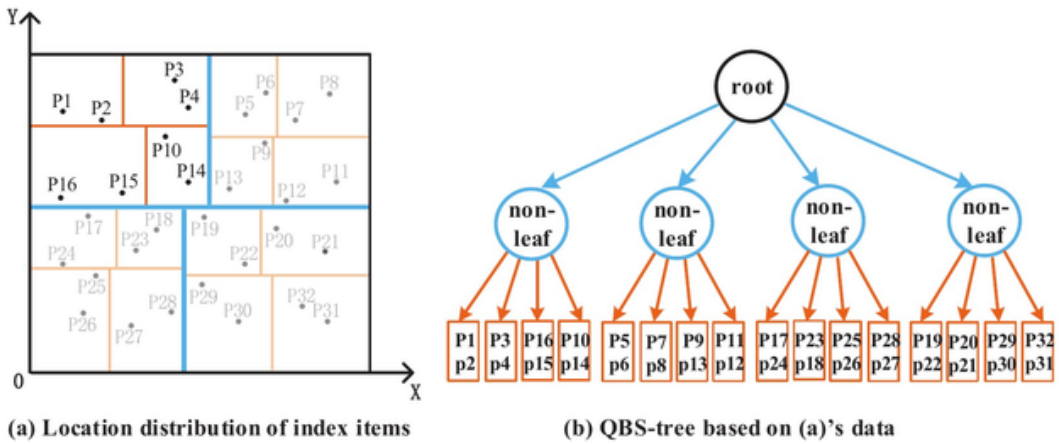
Sort-Tile-Recursive: There are r points, and each node can store n points. Firstly, sort points in ascending order on the y dimension. According to the order, these points are divided into $\sqrt{r/n}$ equal portions and store in $\sqrt{r/n}$ nodes. Secondly, like the first step, points in each node are sorted in the x

dimension, and then points in each node are divided into $\sqrt{r/n}$ equal portions according to the new order and then stored in $\sqrt{r/n}$ new nodes. Finally, r/n nodes are got, and each node have n points.

The sort operation is used in **Sort-Tile-Recursive**, which is expensive. **Division method** avoids sorting operations through optimization.

Division method: **Division method** only splits r points into four parts by Algorithm 1. In *BinaryDevide*(***S***, *"d", p)*, ***S*** is a set of two-dimensional points. *BinaryDevide* splits ***S*** into two sets on d-dimension. The difference between the sizes of the two sets is determined by p. Within *BinaryDevide*, the quick sort algorithm will be called several times, and only one iteration of the algorithm is performed per call. *BinaryDevide* ends when the selected *pivot* is positioned between the $\left\lfloor S.size\left(p-1/2p\right)\right\rfloor \text{th}$ and $\left\lfloor S.size\left(p+1/2p\right)\right\rfloor \text{th}$ of ***S*** after an iteration. In this way, ***S*** can be split into two sets. *BinaryDevide* doesn't divide ***S***'s elements into two equal parts. Because the *pivot* is randomly chosen, the probability that the *pivot* is just midpoint is only $1/S.size$ and after the *pivot* is chosen, the algorithm needs to traverse ***S***'s elements. To reduce the times of traversal, *BinaryDevide* terminates itself when *pivot* is positioned between the $\left\lfloor S.size\left(p-1/2p\right)\right\rfloor \text{th}$ and $\left\lfloor S.size\left(p+1/2p\right)\right\rfloor \text{th}$ of ***S***. Algorithm 1 eventually divides ***S*** into four parts.

Figure 3.
A demonstration of QBS-tree



(a) Location distribution of index items

(b) QBS-tree based on (a)'s data

Table 2.
Algorithm 1: Split(l)

Input: l, the leaf node need to be split. ***Output***: r, the root node of the new subtree generated by splitting the node l.
1 $\mathbf{S_0}$, $\mathbf{S_1}$ = BinaryDevide($\mathbf{S_l}$, "x", p);
2 $\mathbf{S_{00}}$, $\mathbf{S_{01}}$ = BinaryDevide($\mathbf{S_0}$, "y", p);
3 $\mathbf{S_{10}}$, $\mathbf{S_{11}}$ = BinaryDevide($\mathbf{S_1}$, "y", p);
4 Initialize a nod r;
5 $S_{r[0]} = S_{00}$, $S_{r[1]} = S_{01}$, $S_{r[2]} = S_{10}$, $S_{r[3]} = S_{11}$;
6 return r;

The division takes into account both dimensions and tries to make the divided regions closer to squares. The closer the sub-region is to the square, the better the query performance. The reason is as follows. When querying, the query algorithm needs to go through a path from the root node to a leaf node due to all index items being in leaf nodes. This path is the so-called query path. If the sub-regions are mostly long rectangles, the query rectangle will intersect with more sub-regions. Therefore, the closer the sub-regions are to square, the fewer possible query paths are.

QBS-tree is more suitable for indexing points because each index entry belongs to only one particular sub-region. As for non-point index items, the center point of an index item's MBR can be used to represent the index item. Therefore, each node of QBS-tree needs to record two rectangles, one is the sub-region corresponding to this node, and the other is the MBR of all index items.

The Value of M/m

When the number of index items in a leaf node reaches *M*, the leaf node should be split into four new leaf nodes by Algorithm 1. *M* can be 4 times *m* as the number of items in each new leaf node must be bigger than *m*. But Algorithm 1 doesn't split *l* into 4 nodes equally. So, the number of items in one of the new leaf nodes may be smaller than *m*. The author specifies that *M* is 5 times *m*. After specifying this, *p* needs to satisfy the following conditions: $\left\lfloor M\left(\left(p-1\right)/2p\right)\right\rfloor\left(p-1\right)/2p\geq m$. Another reason the author specifies these multiple relationships is the new leaf nodes don't immediately enter the easy-to-adjust state after splitting. If the number of items in a new leaf node is close to *m* or *M*, the number is likely to decrease to *m* or increase to *M*. This will soon cause QBS-tree to adjust again.

Configurable Balance Factor

Most extant tree spatial indexes are strictly balanced. It means all leaf nodes are in the bottom layer and can guarantee the shortest average query path. However, if there are many updates, these trees tend to lose balance easily, which leads to frequent rebalance. The rebalance of these trees is a very time-consuming process. However, the balance factor of QBS-tree is configurable. If the balance factor is too large, the query path may belong. Therefore, QBS-tree makes the balance factor a parameter to be configured by users.

Insertion Algorithm of QBS-tree

For simplification, Algorithm 2 masks the related operations of *root*. The first line calls the function *ChooseLeafNode(r, e)* to find a *r*'s leaf node that is suitable for inserting *e*, which picks a path from *root* to a leaf node. When it runs to a non-leaf node *n*, the center of *e*'s MBR belongs to which one region managed by someone *n*'s child node, and it turns to which one child node to continue picking. It returns the current node when it executes to a leaf node. At line 3, Algorithm 2 needs to adjust QBS-tree's structure when the condition is True. At line 4, *UnbalancedNode(l)* is called to find the first unbalanced non-leaf node among the upper nodes of *l* after splitting *l*. At line 6, *Split(l)* can be safely called to split *l* due to there is no unbalanced node. When *n* is not *null*, QBS-tree needs to be adjusted. Algorithm 2 can directly rebuild *n* by calling Algorithm 3. Algorithm 3 reconstructed *n* and return the root of the new sub-tree. At the second line of Algorithm 3, all elements in *n* are aggregated into one leaf node *l* by calling *AggregateToleaf(n)*. Algorithm 3 redistributes the elements in *n* by Algorithm 1.

Assaying above, if Algorithm 2 directly rebuild *n*, the depth *D* of *n* will be in the range that is shown at Formula (3). The implication of symbols in Formula (1), (2) and (3) is as follows: *N* is the number of index items in *n*; *d* and *M* is the same as that in Table 1; *p* in Formula (2)(3) and *p* in Algorithm 1 are the same. If the index items are evenly divided by Algorithm 3, the number of index items in each new leaf is equal. *D* is the largest integer satisfying Formula (1), which is the smallest one. If the elements are divided in the most uneven way, each time *BinaryDevide* is called, *pivot* is

Table 3.
Algorithm 2: Insertion(r, e)

Input: *r*, the root node of QBS-tree; *e*, the element need to be inserted.
1 l = ChooseLeafNode(r, e);
2 add e to $\mathbf{S}_l$;
3 **if** $S_l.size > M$ **then**
4 n = UnbalancedNode(l)
5 **if** *n* is *null* **then**
6 *r* = *Split(n)*;
7 **else**
8 m = MinUnbalancedNode(n);
9 Redistribution(m);

at the $\left\lfloor \mathrm{S}_n.size\left(\left(p-1\right)/2p\right)\right\rfloor \mathrm{th}$ or $\left\lfloor \mathrm{S}_n.size\left(\left(p+1\right)/2p\right)\right\rfloor \mathrm{th}$. *D* is the largest integer satisfying Formula (2), which is the biggest one. Finally, Formula (3) is deduced from Formula (1) and (2).

Some values among the range of Formula (3) may also cause QBS-tree to imbalance. So, Algorithm 2 need to find the smallest subtree *m* to rebuild, and QBS-tree must be balanced after rebuilding. *MinUnbalancedNode(n)* implements this function. It calculates the depth range of *n* with Formula (3). If someone's depth in the range causes an imbalance in *n*'s higher-level nodes, *MinUnbalancedNode* is recursively called with the higher-level node as a parameter. Finally, *MinUnbalancedNode(n)* will find a subtree that doesn't cause QBS-tree to be imbalanceed after rebuilding. Finally, at line 9, *m* can be safely rebuilt.

$$N\left(\frac{1}{2^d}\right)^D < M \tag{1}$$

$$N\left(\frac{p+1}{2p}\right)^{dD} < M \tag{2}$$

$$\left(1\right),\left(2\right) \rightarrow \left\lfloor \log_{2^d} \frac{N}{M} \right\rfloor \leq D \leq \left\lfloor d \log_{\frac{2p}{p+1}} \frac{N}{M} \right\rfloor \tag{3}$$

Deletion Algorithm of QBS-tree

Deletion algorithm is shown in Algorithm 4. The first line calls *FindLeaf(r, e)* to find the leaf node *l* in where *e* store. At line 4, if the condition is True, Algorithm 4 need to execute the code from line 5 to line 13 to adjust the structure of QBS-tree. If the fifth line is True, Algorithm 4 can rebuild *l.pa* so that the number of index items in all leaf nodes doesn't cross the two boundaries. But this may lead to a *l.pa*'s upper node to out of balance. As with the 8th and 9th line in Algorithm 2, Algorithm 4 need to find the smallest subtree to rebuild. If the fifth line is False, Algorithm 4 combine all the child nodes of *l.pa* into one leaf node at line 9. But this may also lead to an upper node to out of balance. As with lines 4, 8, and 9 in Algorithm 2, Algorithm 4 need to find the earliest imbalance node *n* and the smallest subtree *m* that can be rebuilt and then reconstruct *m*.

Table 4.
Algorithm 3: Redistribution(n)

Input: *n*, the node need to be redistribute. ***Output***: *r*, The root node of the new subtree generated by redistributing *n*.
1 **if** *n* is non-leaf node **then**
2 l = AggregateToleaf(n);
3 return *Redistribution(l)*;
4 **else**
5 r = Split(n);
6 **for** *i* = 0 to 4 **do**
7 **if** $\boldsymbol{S}_{r[i]}.size$ > M **then**
8 Redistribution(r[i]);
9 return *r*;

GLOBAL INDEX

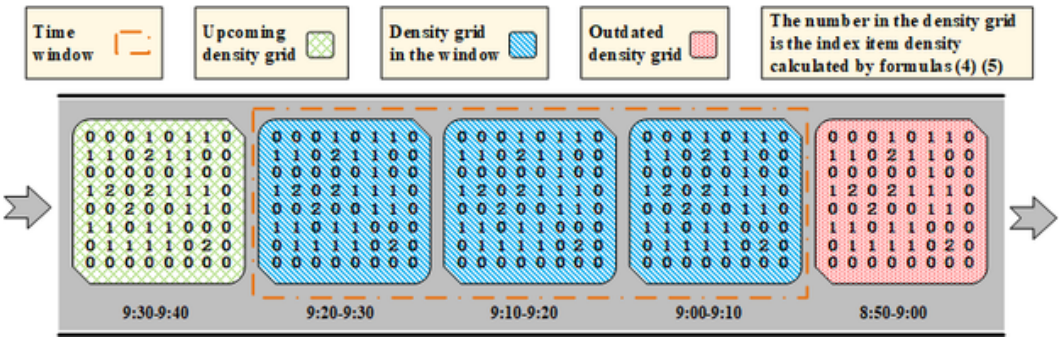
In Figure 2, GQ-tree is responsible for dividing the global region into several sub-regions. The features of GQ-tree are as follows.

1. Each non-leaf node divides its corresponding region into four disjoint sub-regions;
2. Any leaf node doesn't store index items, which only represents a sub-region. But, the number of index items whose center point is in this sub-region will be stored;
3. The number of index items in any leaf node has an upper limit *M* and a lower limit *m*, and *M* is twice *m*;

Table 5.
Algorithm 4: Deletion(r, e)

Input: *r*, the root node of QBS-tree. *e*, the element need to be deleted.
1 l = FindLeaf(r, e);
2 **f** *l* is not *null* **then**
3 remove *e* from $\boldsymbol{S}_l$;
4 **if** the size of $\boldsymbol{S}_l$ is less than *m* **then**
5 **if** the size of $\boldsymbol{S}_{l.pa}$ is bigger than *M* **then**
6 m = MinUnbalancedNode(l.pa);
7 *Redistribution(m)*;
8 **else**
9 initialize a new leaf node *nl* to replace *l.pa*, and add all the elements in *l.pa* to $\boldsymbol{S}_{nl}$;
10 n = UnbalancedNode(nl);
11 **if** *n* is not *null* **then**
12 m = MinUnbalancedNode(n);
13 *Redistribution(m)*;

Figure 4.
Density grid queue



4. The tree height of GQ-tree is at least two.

Density Grid

Maintaining GQ-tree needs to know the spatial distribution of all index items. However, the input tuples only flow through GU but don't store in it. The author's solution is that maintaining a density grid in GU to represent the spatial distribution. A density grid is a 2D array. The density grid divides the global region into *m*n* equal-sized regions. An element of the array represents the number of index items in the corresponding grid.

Formula (4)(5) can calculate the grid to which an index item belongs. The global region can be represented by a rectangle $\left[\left(Glow_x, Glow_y\right), \left(Ghigh_x, Ghigh_y\right)\right]$. $Glow_x$ and $Glow_y$ are the x-axis and y-axis coordinate values of the global rectangle's left-bottom point, respectively. $Ghigh_x$ and $Ghigh_y$ are the x-axis and y-axis coordinate values of the top-right point, respectively. The global rectangle is divided into grids of *m* rows and *n* columns. There is an index item flow through GU, and its center point is *(x, y)*. The system should increment *array*[*row*][*col*] by one.

$$row = \left\lfloor \left(x - Glow_x\right) \Big/ \left(\left(Ghigh_x - Glow_x\right)/m\right) \right\rfloor \tag{4}$$

$$col = \left\lfloor \left(y - Glow_y\right) \Big/ \left(\left(Ghigh_y - Glow_y\right)/n\right) \right\rfloor \tag{5}$$

This paper solves Real-Time Range Query, so it only focuses on the data generated in the last *n* minutes. The fact is there are not only tuples flowing into the window, but also tuples flowing out, in Figure 1. Therefore, GU should also remove the outdated tuple from the density grid. The author uses a density grid queue to solve this problem, in Figure 4. The rightmost density grid represents the distribution of the index items generated from 8:50 to 9:00. When all index items generated from 8:50 to 9:00 slide out of the window, the density grid in use should be reduced by 8:50 to 9:00 density grid. In this way, GU can know the spatial distribution of all index items in the time window.

Update GQ-tree

The sub-regions divided by GB-tree will slowly produce data skew. Therefore, GU needs to regularly adjust GQ-tree structure to ensure the number of index items between sub-regions is approximately

Table 6.
Algorithm 5: GetAdjustNodes(leafNodes)

Input: *leafNodes*, all leaf nodes of the global index. ***Output***: *nodes*, all sub-tree of the global index that needs to be reconstructed.
1 initialize a list *nodes* to store all sub-tree that needs to be reconstructed;
2 **for** *l* in *leafNodes* **do**
3 **if** $S_l.size > M$ || ($S_l.size<m$ && *l* isn't any child of *root* when *root*'s children are all leaves) **then**
4 **if** $S_l.size < 4.5*m$ **then**
5 node = l.pa;
6 **if** !($S_{node}.size >= m$ && $S_{node}.size <= M$) **then**
7 **while** $S_{node}.size < 4.5*m$ && *node* isn't *root* **do**
8 *node = node.pa*;
9 **else**
10 node = l;
11 **if** *node* is not a descendant node of any node in *nodes* **then**
12 Remove the descendant nodes of *node* in *nodes*;
13 add *node* to *nodes*;
14 return *nodes*

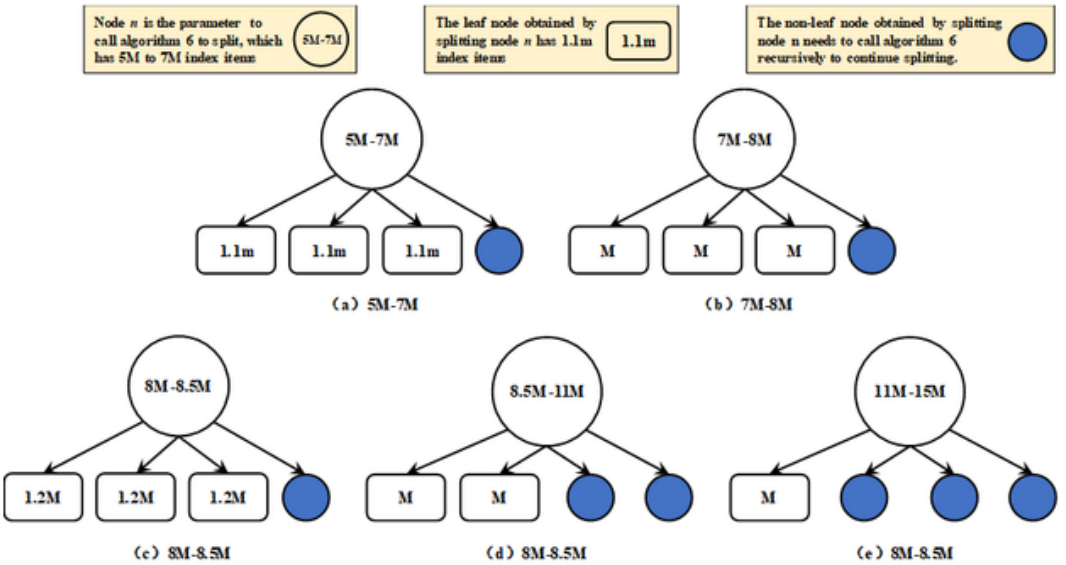
equal. GU update GQ-tree in the following two steps. Firstly, to get all subtree that needs to be reconstructed, Algorithm 5 is called with all leaf nodes as a parameter. Secondly, Algorithm 6 is called to rebuild these subtrees.

At line 3 of Algorithm 5, if the number of index items in a leaf node isn't between *M* and *m*, it means GB-tree needs to reconstruct someone subtree. But, when the four child nodes of *root* are all leaf nodes, GB-tree ignores the index item number of any leaf node is lower than *m*. At line 4, if the condition is False, it means that the leaf node can be split directly. Otherwise, GB-tree needs to find the smallest subtree that can be reconstructed, using line 5 to 8. If a leaf node can be split directly, its index item number must be greater than 4**m*. There are no index items in GU, so that a leaf node cannot be equally divided into four child nodes only with a density grid. It's why Algorithm 5 set the value to 4.5**m* instead of 4**m* at line 4. Line 6 means that GB-tree can't merge *l* and all of *l*'s brother node into one leaf node. So, Algorithm 5 should use line 7 to 8 to find the smallest subtree to be reconstructed. Similar to line 4, the reconstructed subtree needs to meet the splitting conditions that the number of index items must be more than 4.5**m*, in addition to the root. Line 11 to 12 guarantees that any element of *nodes* isn't a descendant node of any other element.

Algorithm 6 is a recursive function and calls itself and *fourSplit*(*size1, size2, size3*) to reconstruct subtree *n*. *fourSplit* firstly finds a boundary on the y-axis to divide the region corresponding to *n* into upper and lower parts, ensuring that the number of index items contained in the upper sub-region has exceeded *size1* plus *size2*. Then, *fourSplit* finds a boundary on the x-axis to divide the upper sub-region into two left and right sub-regions, so that the number of index items in the upper left sub-region just exceeds *size1*. Similarly, the function finds an x-axis bound in the lower sub-region so that the number of index items in the lower left sub-region is greater than *size3*. Finally, *fourSplit* use four new leaf nodes to store the index items in the four sub-regions, respectively.

Algorithm 6 describes how sub-regions are divided when $S_n.size$ and *Mid* are in different quantitative relationships. From line 2 to 5, as long as $S_n.size$ is bigger than 15**Mid*, the index items of *n* is divided into four parts by calling *fourSplit*, and then the four child nodes of *n* recursively call

Figure 5.
Different split strategies



Algorithm 6 for splitting. As the recursion progresses, $S_n.size$ will eventually fall into the range below line 6. At line 6, if the condition is True, it means that Algorithm 6 has separated a qualified leaf node whose index items are between *M* and *m*. At line 8 to 9, Algorithm 6 can call *fourSplit* to split *n* into four qualified leaf nodes. When the value of $S_n.size$ meets any condition of the code branch below line 11, the division of *n* can be represented by Figure 5, and M is equivalent to *Mid* in Figure 5. For example, Figure 5-(b) corresponds to line 13 to 14 of Algorithm 6. The first three child nodes of *n* are directly divided into leaf nodes, and each of them has about *Mid* index items. Then, Algorithm 6 recursively calls *AdjustNode* with the fourth child as the parameter to split it.

IMPLEMENT GQ-QBS BASED ON APACHE FLINK

Apache Flink Overview

Flink is a popular datastream processing framework and often used for real-time computing. The user submits Flink job to JobManager, and JobManager parses a job into a Directed Acyclic Graph (DAG). The vertex of DAG is a calculation task, which receives data and then completes calculation logic and finally outputs the calculated result. The line of DAG represents the transmission of data. After JobManager parses a job, the tasks are distributed to TaskManagers to executing. TaskManager is a JVM process. Tasks run in slots, and slot is a thread of TaskManager. TaskManagers is responsible for scheduling tasks and communicating with JobManager and other TaskManagers.

Figure 6 is an example, which has three calculation tasks Density Grid Task (DGT), Global Task (GT) and Local Task (LT). A task can have multiple degrees of parallelism; for example, GT's parallelism is three, and each of them is called a subtask. Different subtasks of the same task are distributed in different slots. But Flink allows multiple subtasks to share a slot, as long as these subtasks belong to the same job.

Table 7.
Algorithm 6: AdjustNode(n)

Input: *n*, the node that needs to be reconstructed. ***Output***: *r*, the root of the reconstructed subtree.
1 *Mid* = *(m + M)*/2;
2 **if** S_n.*size* >= 15**Mid* **then**
3 r = fourSplit($\mathbf{S}_n$.size/4, $\mathbf{S}_n$.size/4, $\mathbf{S}_n$.size/4);
4 **for** *i* = 0 to 4 **do**
5 *r*[*i*] = *AdjustNode*(*r*[*i*]);
6 **else if** S_n.*size* >= *m* && S_n.*size* <= *M* **then**
7 return *n*;
8 **else if** S_n.*size* >= 3**Mid* && S_n.*size* <= 5**Mid* **then**
9 r = fourSplit($\mathbf{S}_n$.size/4, $\mathbf{S}_n$.size/4, $\mathbf{S}_n$.size/4);
10 **else**
11 **if** S_n.*size* >= 5**Mid* && S_n.*size* <= 7**Mid* **then**
12 *r* = *fourSplit*(1.1**m*, 1.1**m*, 1.1**m*);
13 **else if** S_n.*size* >= 7**Mid* && S_n.*size* <= 8**Mid* **then**
14 *r* = *fourSplit*(*Mid*, *Mid*, *Mid*);
15 **else if** S_n.*size* >= 8**Mid* && S_n.*size* <= 8.5**Mid* **then**
16 *r* = *fourSplit*(1.2**m*, 1.2**m*, 1.2**m*);
17 **else if** S_n.*size* >= 8.5**Mid* && S_n.*size* < 11**Mid* **then**
18 *r* = *fourSplit*(*Mid*, *Mid*, (S_n.*size* – 2**Mid*)/2);
19 *r*[2] = *AdjustNode*(*r*[2]);
20 **else**
21 *r* = *fourSplit*(*Mid*, (S_n.*size* – *Mid*)/3, (S_n.*size* – *Mid*)/3);
22 *r*[1] = *AdjustNode*(*r*[1]);
23 *r*[2] = *AdjustNode*(*r*[2]);
24 *r*[3] = *AdjustNode*(*r*[3]);
25 return *r*;

Implement Distributed Index with Flink

Figure 6 demonstrates how to implement Real-time Range Query with Flink by GQ-QBS. The data flow and processing of Figure 6 is as follows. The input of DGT are index or query items. DGT is responsible for computing the density grid of index items, a density grid is calculated for each sliding interval of time window, and then broadcast the density grid to each subtask of GT. Every input item is randomly forwarded to one subtask of GT. The parallelism of GT is three, to ensure the consistency of GQ-tree in the three subtasks, it's necessary to keep the density grids in different subtasks consistent.

After GT's subtask receives an index item or query item, how to forward it to LT's subtask is as follows. Both query items and index items can be represented as a rectangle. The strategy is to send an input tuple to subtasks in which the subregions are corresponding to intersect the rectangle representing the input tuple. Each subtask of LT is responsible for executing query and update. Finally, the Local Index outputs every query result.

When there is a load imbalance between LT's subtasks, GQ-tree can perceive it through the density grid, and then reconstruct a certain subtree. After reconstruction, GQ-tree needs to allocate subtasks of LT to the new leaf nodes and migrate the index items in the old subtasks to the new subtasks. To reduce the amount of data migration, the new leaf node can reuse the previous subtask of LT. For example, if the index region corresponding to a new leaf node and the index region corresponding to an old leaf node is roughly the same and the corresponding index items are also roughly the same, the subtask of LT corresponding to this old leaf node is assigned to this new leaf node. Hungarian algorithm can be used to minimize data migration. After determining the source and purpose of the data to be migrated, the system can use Flink's queryable state mechanism or intermediate storage such as Redies to realize the data migration between LT's subtasks.

Space Efficiency

The global index GQ-tree and the local index QBS-tree that make up the distributed spatial index GQ-QBS are both quad-tree. The number of leaf nodes *l* and that of intermediate nodes *i* of quad-tree satisfy $l+i=4*i+1$. Assuming that QBS-tree has *n* index items, there are probably $n/2.5*m$ leaf nodes. The maximum number of GQ-tree's leaf nodes is determined by the parallelism of LT, which is assumed to be denoted as *p*. The extra space consumed by QBS-tree and GQ-tree is the number of nodes multiplied by the space occupied by one node. From the above information, it can be concluded that the total number of QBS-tree's nodes and GQ-tree's nodes are $\left(8*n-5*m\right)/\left(15*m\right)$ and $\left(4*p-1\right)/3$ respectively.

RELATED WORK

Single-node Spatial Index

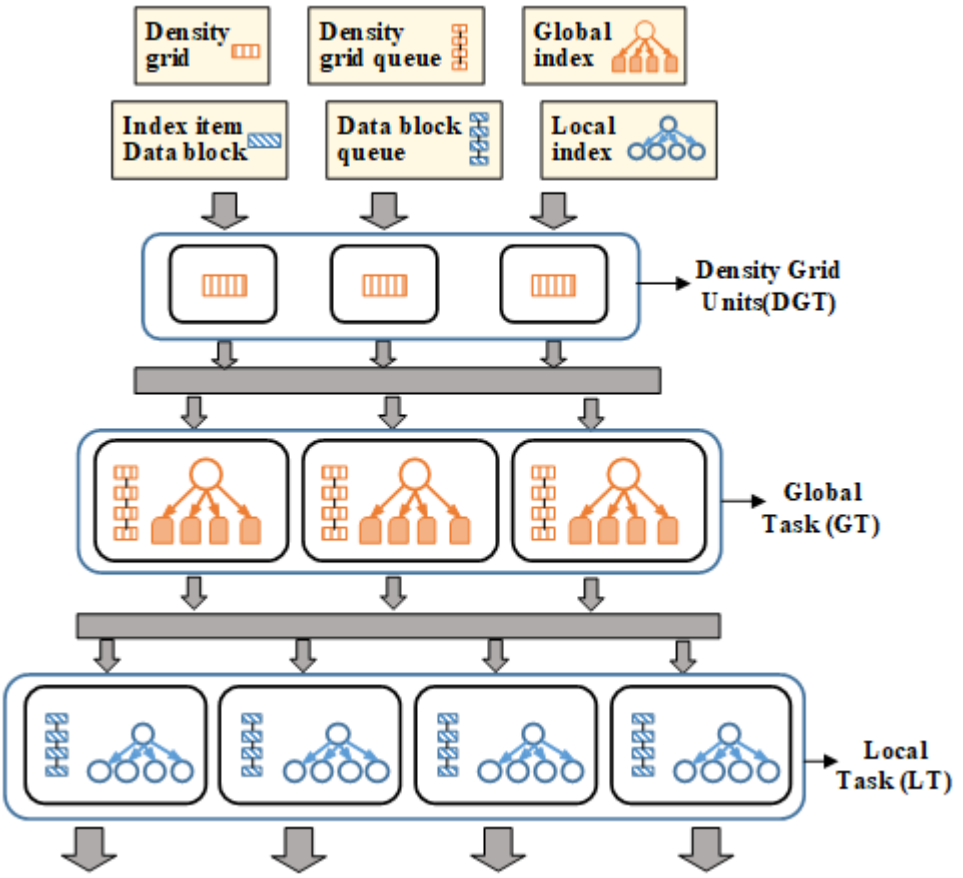
R-tree and its variants are the most commonly used single-node spatial indexes. When indexing a static data set, the query performance of an index should be paid more attention to. There are many factors that affect the query performance of an R-tree, and these factors also affect each other. The paper (Milo & Suciu, 1999) analyzes these factorfs. The two main factors are listed below: 1) the overlap area between the MBRs of the same layer nodes should be as small as possible, which helps to reduce the number of query paths when querying; 2) the smaller the perimeter of node's MBR, the higher the aggregation degree of a tree, the closer the distance between the index items in the same node, the smaller the number of query paths when query.

Hilbert-tree (Kamel et al., 1994) uses a Hilbert curve to sort the index items, and the distance between adjacent index items is very close after sorting. Hilbert-tree built based on this sorted sequence has a relatively small perimeter of node's MBR, and the index items in the same node have a good degree of aggregation. R*-tree (Beckmann et al., 1990) also takes into account the perimeter of node's MBR and the overlap area of MBR between the same layer nodes when building or updating a tree.

STR-tree (Leutenegger et al., 1997) uses its packing algorithm Sort-Tile-Recursive to reduce the overlap area between the MBRs of the same layer nodes. When indexing points, there is no overlapping area between the MBRs of the same layer nodes; when indexing non-point index items, the overlapping area is also tiny. The experiment in the paper (Leutenegger et al., 1997) also verified that the aggregation degree of nodes generated by packing algorithm Sort-Tile-Recursive is better than that of Nearest-X (Roussopoulos et al., 1985) and Hilbert Sort (Kamel et al., 1994).

VP-tree (A. Fu et al., 2000) and KD-tree (Zhou et al., 2008) are both binary trees, and each node of them store an index item. The index item in a node *n* divides all index items in the subtree represented by *n* into the same two parts in a certain dimension. There is only one query path when performing point query in KD-tree and VP-tree, while there are multiple query paths when performing range

Figure 6.
Implement index on Apache Flink



query. The overlap area isn't big between the MBRs of the same layer nodes of KD-tree or VP-tree, so there won't be too many query paths when range query.

QR-tree (Y. Fu et al., 2003) combines R-tree and quadtree to make up for the shortcomings of R-tree, which is too much overlap area between MBRs of the same layer nodes. Each node of QR-tree has an index region, and its four child nodes divide the region into four equal parts; that is, each dimension is equally divided once. Each node has an R-tree, which indexes the index items only belonging to the index region of this node. This significantly reduces the overlap of MBRs of the same layer nodes.

PH-tree (Zaschke et al., 2014) is a very efficient multi-dimensional index. It uses the method of sharing binary number prefixes to build PH-tree, which makes its query and update performance very efficient. But PH-tree is more suitable for point query. When performing range queries, as the query area increases, its query performance drops sharply.

When updates are frequent, the above index is difficult to adapt. They either need to adjust the tree's structure frequently or rebuild the entire tree or a subtree. For example, STR-tree, KD-tree, and VP-tree are static indexes. When they need to be updated, they need to rebuild the entire tree. Both R-tree and Hilbert-tree adopt a bottom-up update method. When the amount of updates is large, they need to adjust the tree structure frequently. Although PH-tree has better update and query performance, when the query rectangle of range query is large, the query performance of PH-tree is poor. The spatial index QBS-tree introduced in this paper makes the tree's balance factor configurable through

a top-down update algorithm, reducing the number of times the tree adjusts its structure. Moreover, when executing wide-range queries on QBS-tree, its query performance won't drop much.

For indexing a large number of moving objects, there is some work (Deng et al., 2015; Sidlauskas et al. 2011; Xiong & Marble, 1996) to improve the query and update performance of spatial index through multi-threaded processing, GPU processing, and other parallel processing methods. The above spatial index for indexing moving objects mainly uses better hardware to speed up query and update performance. In contrast, the single-node spatial index QBS-tree introduced in this paper improves query and update performance by its algorithm.

DISTRIBUTED SPATIAL INDEX

Spatial Index Base on Hadoop

Although QBS-tree can significantly improve the update and query performance, single-node indexes will still produce performance bottlenecks when indexing a sizeable spatial data set. Distributed spatial indexes are often used to avoid single-node performance bottlenecks. A lot of work implements spatial query on the distributed computing framework Hadoop.

Hadoop-GIS (Aji et al., 2013) is a spatial data warehouse system that processes large static spatial data sets. Hadoop-GIS uses a global partition index to realize a spatial-partition distributed spatial query framework. Hadoop-GIS is also integrated into Hive to implement a declarative spatial query function. The performance of Hadoop-GIS is better than traditional spatial relational databases.

SpatialHadoop (Eldawy & Mokbel, 2013) optimizes Hadoop to improve the performance of processing spatial data significantly. SpatialHadoop optimizes Hadoop in the language layer, storage layer, MapReduce layer, and operation layer, and its leading optimization is in the storage layer. It uses a master-slave distribution spatial index to index spatial data set, which includes a global index and several local indexes. The global index partitions data set, and each partition uses a local index to index its data set. When performing a range query, the global index is used to filter out the partitions that won't contribute to the query result. Then the local index is used to execute the query on the remaining partitions.

In addition to the above two indexes, Lu (Lu at al., 2012) and Zhang (Zhang et al., 2009) also use MapReduce to implement range query and spatial join operations. Although the performance of these indexes for processing static spatial data set is better than that of traditional relational databases, these indexes cannot handle dynamic data sets which change frequently.

Spatial Index Base on Hbase

Apache Hbase is a distributed, column-oriented database, and it's also a KeyValue structure database. Using Hbase to build distributed spatial index is also a research direction. Huang (Huang et al., 2014) uses grids to group a spatial data set. Data in the same grid is grouped into a group, and then the grid is indexed by R-tree. When querying, first use R-tree to find the grids in the query area, then use the grids to find all the spatial data in the corresponding group from Hbase, and finally filter those spatial data. Its disadvantage is that the size of the grid is fixed, and the number of index items in different grids may be unbalanced, which will cause too many index items to be selected from Hbase when querying.

Nishimura (Nishimura et al., 2013) use linearization technology to map multi-dimensional data into Hbase row key. However, the spatial data sorted by linearization technology cannot guarantee that all adjacent data are spatially close neighbours. False-positive will occur when using that index to execute a query. To reduce the number of false-positives, Jo (Jo & Jung, 2018) build a multi-dimensional index above Hbase client, called quadrant-based minimum bounding rectangle tree (QbMBR-tree). QbMBR-tree partitions the index items more accurately and uses QbMBR-tree to index the index items.

Although distributed spatial indexes based on Hadoop and Hbase have better performance, they only have advantages when indexing a relatively static spatial data set. In addition, the data and intermediate calculation results of Hadoop and Hbase are stored in hard disk, so these indexes cannot meet the needs of applications with high real-time requirements.

Spatial Index Base on Spark or Storm

Compared to Hadoop, memory-based computing frameworks, such as Spark, Flink, and Storm, can process data faster. Some works implement distributed spatial index on these frameworks. Xie (Xie et al., 2017) implements a distributed spatial index on Apache Spark, and Zhang (Zhang et al., 2016) implements another on Apacha Storm. These two indexes are distributed spatial indexes of master-slave structure.

Zhang (Zhang et al., 2016) sends the index items received in Spout of Storm to the slave node in a polling manner. Each slave node establishes a spatial index base the set of received index items. Zhang (Zhang et al., 2016) broadcasts the query items received from Spout of Storm to all slave nodes, and all queries need to be executed in every slave node. The union of the query results of a query item in all slave nodes is the final result. The advantage of this distributed spatial index is that there will be no load imbalance as the dynamic spatial-temporal data set changes. The disadvantage is that each query item has to execute a query in all slave nodes, which consumes many resources.

Xie (Xie et al., 2017) uses Sort-Tile-Recursive packaging algorithm to build an STR-tree on the master node. This tree does not store index items, and it only divides space. Each of its leaf nodes corresponds to an index area, the index areas don't intersect, and each slave node corresponds to an index area. The master node receives the query items and index items and sends them to the corresponding slave nodes. The slave node establishes a single-node spatial index on the set of index items and processes the query items. This paper refers to the partitioning method of this distributed spatial index as the fixed area partitioning method. The advantage of this distributed spatial index is that query items only need to be sent to some slave nodes to execute the query, which consumes fewer resources. The disadvantage is that as the index is updated, there may be load imbalance among slave nodes.

EXPERIMENT

Experimental Setup

Environment: All experiments are performed on a high-performance server, and its hardware parameters are as follows: CPU: 128*Intel(R) Xeon(R) CPU E7-8860 v3 @ 2.20GHz, Memory: 2TB @1600 MHz. A Flink cluster is deployed on this server, and its configuration is as follows: 128*slot, the memory of JVM is 985MB, and the memory of outside JVM is 3289 MB. All algorithms are implemented in Java.

Data Sets: The experiment used two real-world data sets.

DIDI-CD: This data set is the trajectory data from Didi Chuxing (https://gaia.didichuxing.com). The coordinate range of this trajectory set is a certain area of Chengdu, China. Its size is 900GB. Each piece of data includes a taxi ID, a sampling timestamp and a position (latitude and longitude).

TAXI-BJ: This data set is the trajectory data of some taxis in Beijing, China in 2008. It contains about one million entries, and the structure of the entry is similar to DIDI-CD. The distribution of TAXI-BJ is even more uneven than DIDI-CD.

Performance Metrics: This paper uses the following metrics to evaluate the performance of an index. *a) time* is the time required to perform several updates or queries operations. *b) delay* is the time interval from input to output when processing data with Flink. *c) throughput* is the reciprocal of the time required to process a static dataset with Flink.

Performance Parameters: In this paper, the comparative experiment is performed by adjusting the following parameters. *a) size* is the approximate number of index items indexed in a spatial index. *b) radius* represents the size of the square to perform a range query, and *2*radius* is the side length of a range query square. *c) ratio* is the ratio of the number of queries to that of updates. In Section EXPERIMENT, one update includes one insert and one delete.

Comparison of Single Node Indexes

This part compares the performance of different spatial indexes when indexing points and rectangles separately. The author uses the minimum bounding rectangle of a segment composed of adjacent sampling points of a track as the index rectangle. During execution, the index items will be inserted into the index until the number of index items in the index reaches *size*. After that, each insertion will be accompanied by deletion to keep the number of index items in the index unchanged. And at the same time, the program started to count the *time* consumed, which is used as the performance metric, for 20,000 updates and 20,000**ratio* queries when indexing points, or 3000 updates and 3000**ratio* queries when indexing rectangles.

In the experiments corresponding to Figure 7 and Figure 8, *ratio* is 2 and *size* is 40,000. Among these experiments, QBS-tree takes the least time to complete the specified operations, and as *radius* increases, the growth rate of its curve is the slowest. Especially QBS-tree performs better when indexing the more unevenly distributed data set TAXI-BJ. This is because QBS-tree is a quadtree, and there is little overlap between MBRs of nodes at the same level, and there are fewer query paths when QBS-tree executes a query. Another important reason is that QBS-tree is updated quickly, which also makes it less time-consuming to perform all operations. PH-tree is better at point query and small range query. However, when *radius* becomes larger, the performance of PH-tree will deteriorate. So, the curve corresponding PH-tree grows fast. As *radius* becomes larger, the more partitions the query rectangle spans when performing a query on QR-tree, the more R-trees perform the query in QR-tree. Therefore, the QR-tree curve grows fast. When indexing the unevenly distributed data set TAXI-BJ, the range of QR-tree's partitions is bigger. Therefore, when *radius* increases, the curve corresponding to QR-tree doesn't grow very quickly. Another finding is, when indexing the unevenly distributed data set, the index items in QR-tree are more concentrated in some R-trees, and the performance of QR-tree will gradually degrade to the original R-tree.

In the experiments corresponding to Figure 9 and Figure 10, *radius* is 60 and *size* is 40,000. When indexing the unevenly distributed data set TAXI-BJ, as *ratio* increases, the performance of QBS-tree is the most stable and fastest index. This shows QBS-tree's query performance is also better. When *ratio* goes from three to four and indexing DIDI-CD's rectangle index items, the curves corresponding to all indexes have large fluctuations, which just reflects the uncertainty of the real-world data. But in this uncertain environment, QBS-tree is still the best performer. The rapid growth of the curve corresponding to PH-tree indicates that the query performance of PH-tree is poor. The design concept of QR-tree is the same as that of QBS-tree. They both use a quad node to divide the space in each dimension, so the growth rate of the curve corresponding to QR-tree is similar to that of QBS-tree. But, when indexing the unevenly distributed data set TAXI-BJ, QR-tree gradually degrade to the original R-tree. So, its curve is closer to that of R-tree.

In the experiments corresponding to Figure 11 and Figure 12, *radius* is 100 and *ratio* is 2. And in the experiments corresponding to Figure 13 and Figure 14, *radius* is 60 and *ratio* is 2. When indexing the evenly distributed data set DIDI-CD's points, QBS-tree is better than PH-tree in most cases. With the increase of *size*, there are more index items with repeated positions and index items with the same position occupy only one index position in the PH-tree. So there are fewer nodes in the PH-tree, and query and update are faster. This is why PH-tree's performance in Figure 19 is better than QBS-tree after *size* exceed 80,000. However, QBS-tree's performance is the better one when indexing the unevenly distributed data set TAXI-BJ. And when indexing rectangle, the possibility of index item positions overlapping is smaller, so the performance of QBS-tree is better.

Figure 7.
Radius as variable and index DIDI-CD's rectangle

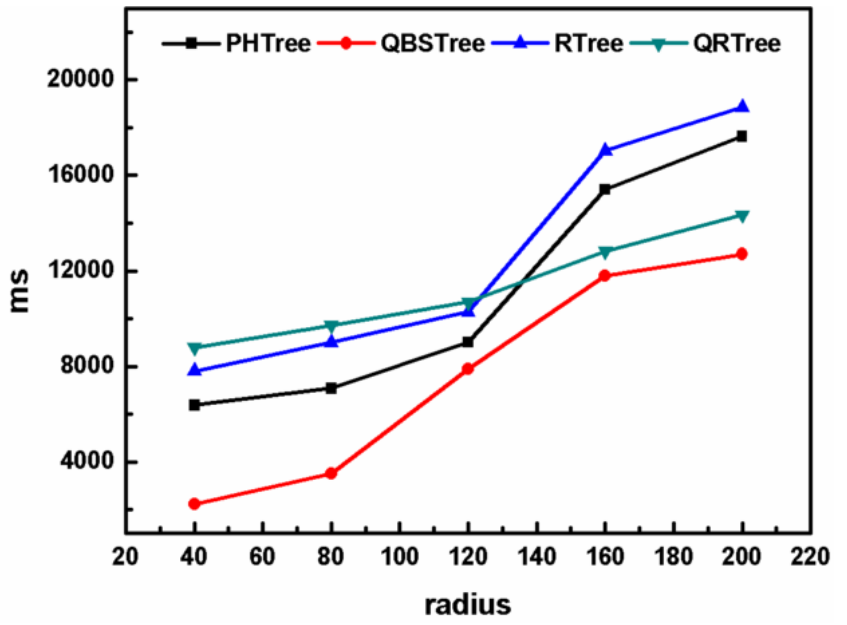
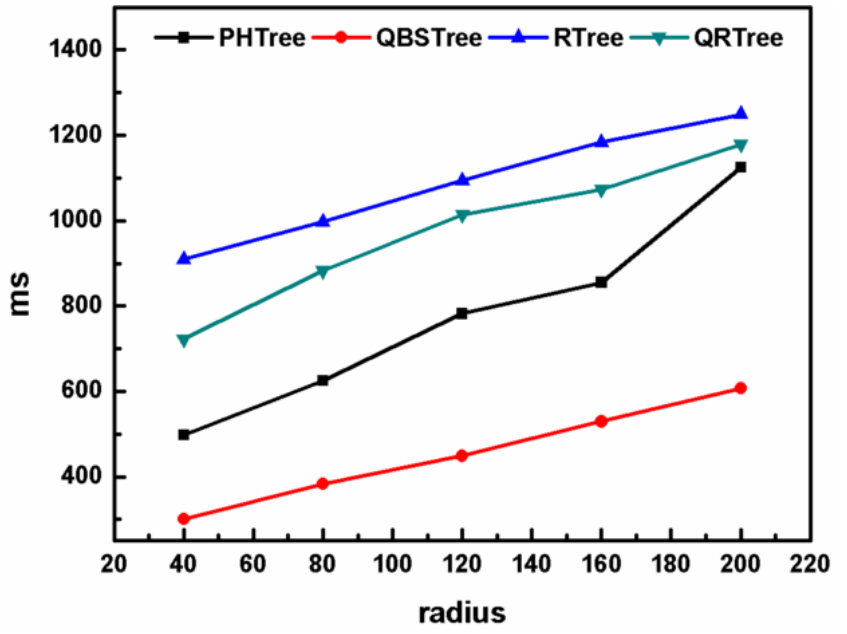


Figure 8.
Radius as variable and index TAXI-BJ's rectangle



In summary, under frequent update scenarios, QBS-tree can show better performance. QBS-tree is especially good at indexing the unevenly distributed data set. The performance of QBS-tree is also better when performing large range queries.

Figure 9.
Ratio as variable and index DIDI-CD's rectangle

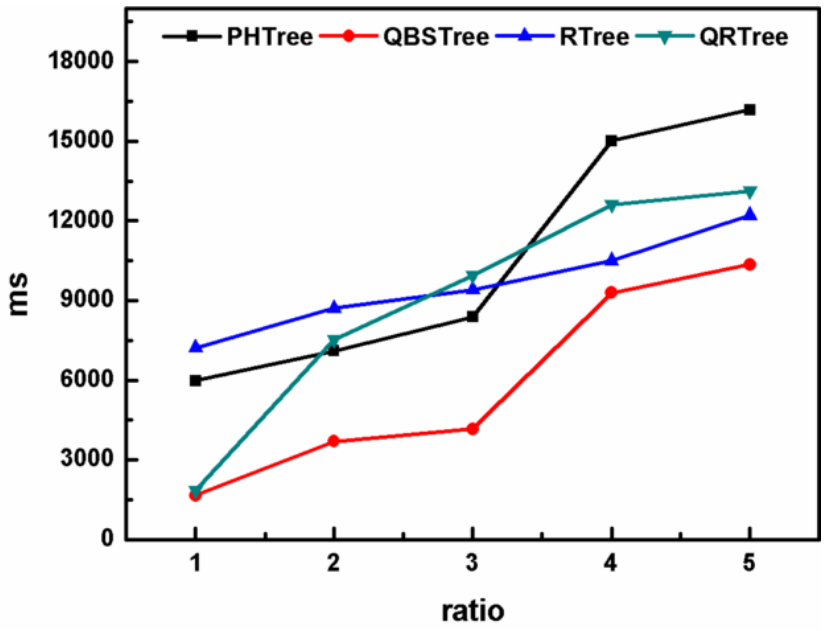
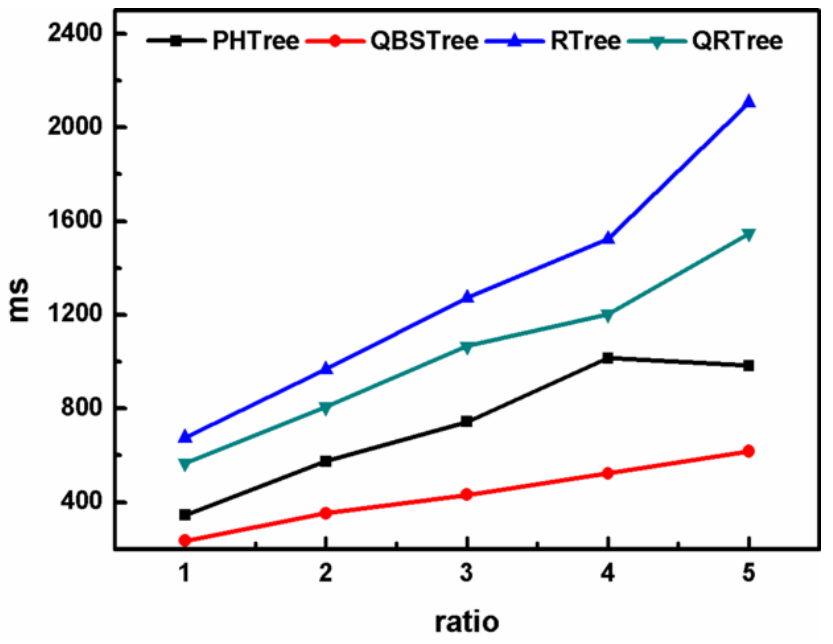


Figure 10.
Ratio as variable and index TAXI-BJ's rectangle



Comparison of Distributed Indexes

QBS-tree is also suitable for a global index. It recursively divides a global region into several sub-

Figure 11.
Size as variable and index DIDI-CD's point

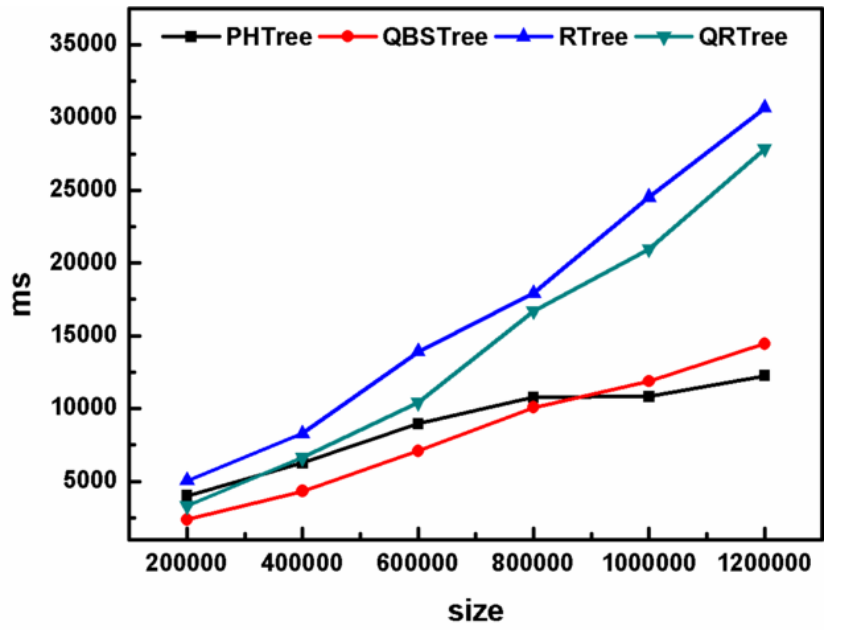
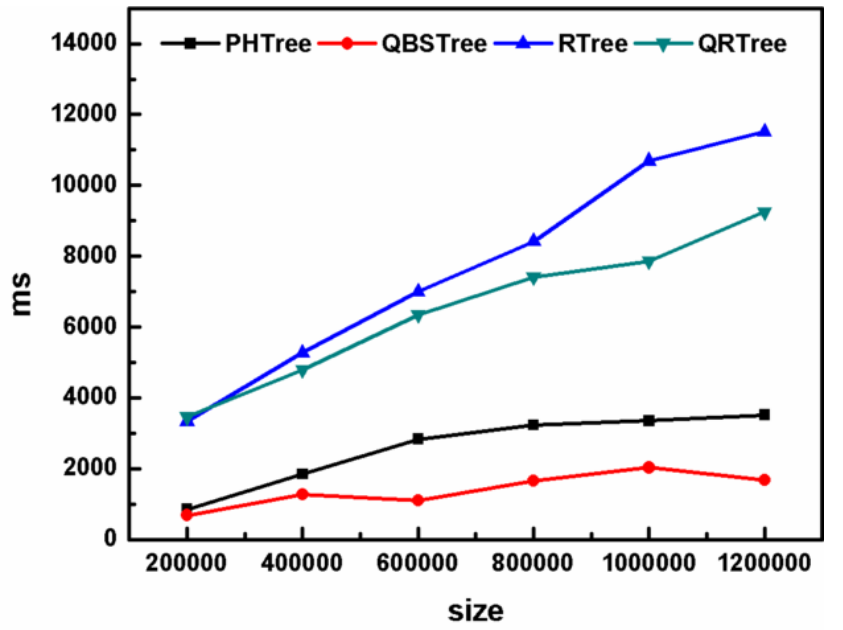


Figure 12.
Size as variable and index TAXI-BJ's point



regions from top to bottom. However, QBS-tree has one drawback, which is that *M* is five times *m*. It means that the total number of index items in a local index is five times that of another local index

Figure 13.
Size as variable and index DIDI-CD's rectangle

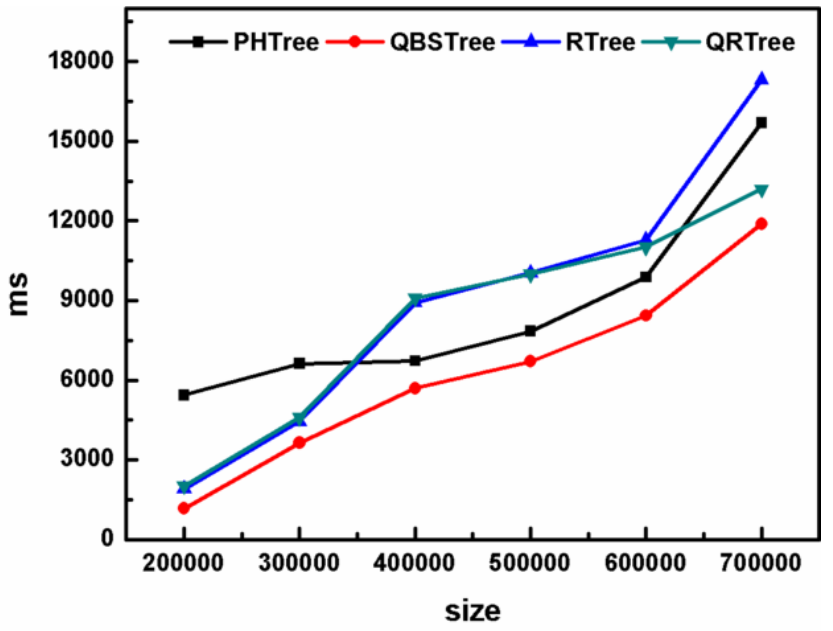
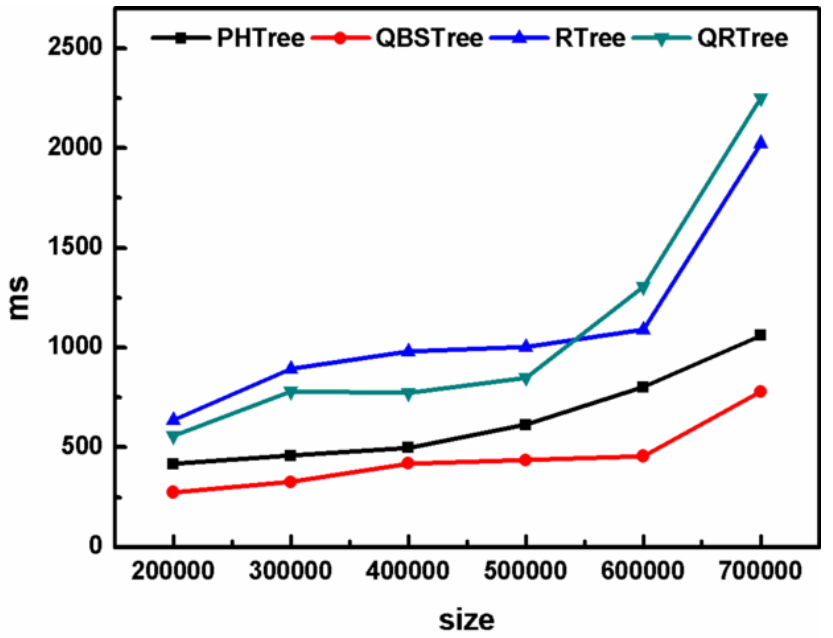


Figure 14.
Size as variable and index TAXI-BJ's rectangle



in the most extreme case. But GQ-tree's *M* is twice as large as *m*. So, GQ-tree can provide better load balancing (Cybenko et al., 1989; Fang et al., 2019) than QBS-tree. Figure 15 and Figure 16 are the

data skew caused by GQ-tree and QBS-tree, respectively. In this contrast experiment, the author kept about 2 million index items in the time window. The ordinate is the ratio of the total number of Local Index's index items to *m*. The upper bound of the box is the value at the $\left(9/10\right)\mathrm{th}$ after sorting, and the lower bound is the value at the $\left(1/10\right)\mathrm{th}$. As *Mid* grows from 60,000 to 300,000, the amount of tuples in each local index is roughly between *m* and $2 * m$ when GB-tree is global index, while the amount is roughly between $1.2 * m$ and $4.8 * m$ when QBS-tree is global index. Therefore, GQ-tree can achieve better load balancing.

When GQ-tree is used as the global index, Figure 17-19 are the performance for the total number of index items at one million, two million, and three million, respectively. The throughput and latency of this distributed index do not fluctuate greatly with changes in the total number of index items. They are mainly determined by *m* of the global index. In other words, the total number of index items in the local index determines the performance of this distributed index.

CONCLUSION

The timeliness of Spatial-temporal data is vital, so those new and outdated index items need to be inserted into or removed from the index in a timely manner. However, the existing spatial indexes are challenging to meet the requirements of real-time systems for update performance. Because fast-changing index items cause these indexes to adjust their structure or rebuild the tree frequently, it results in high maintenance costs, such as HiIndex (Liu et al., 2021), QRB-tree (Yu et al., 2021), SPRIG (Zhang et al., 2021), PH-tree (Zaschke et al., 2014), QSF-Trees (Orlandic, R., & Yu, B, 2004), dynamic grid file (Hou et al., 2009). Some research speed up the performance of index by improving resource utilization, such as increasing parallelism (Song et al., 2004; Choy et al., 2000), but these researches didn't optimize the update algorithm, therefore, these researches can't save computing resources.

Figure 15.
The data skew of GQ-tree

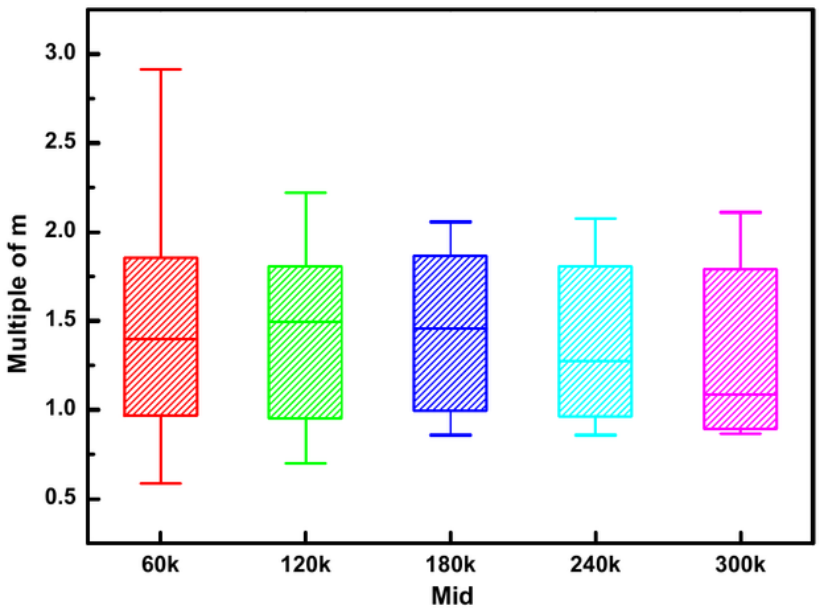


Figure 16.
The data skew of QBS-tree

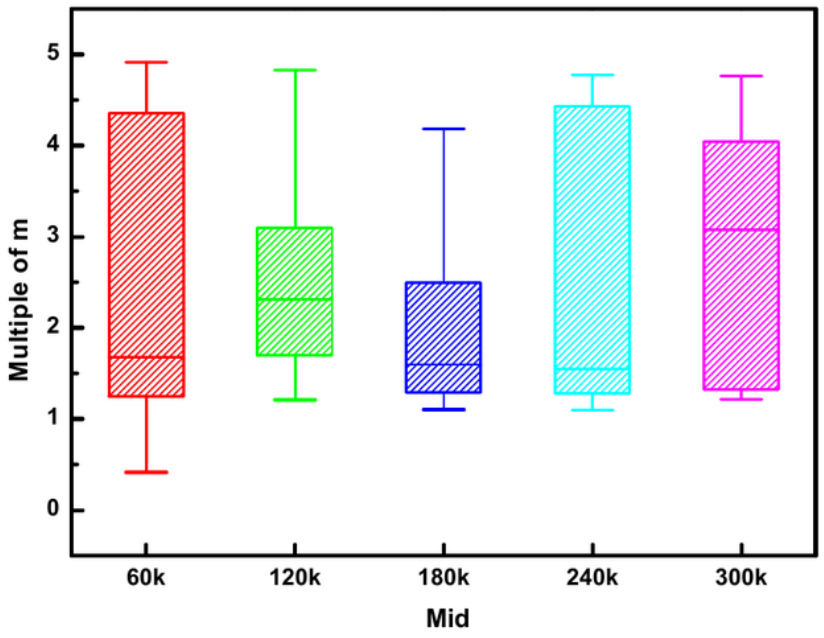
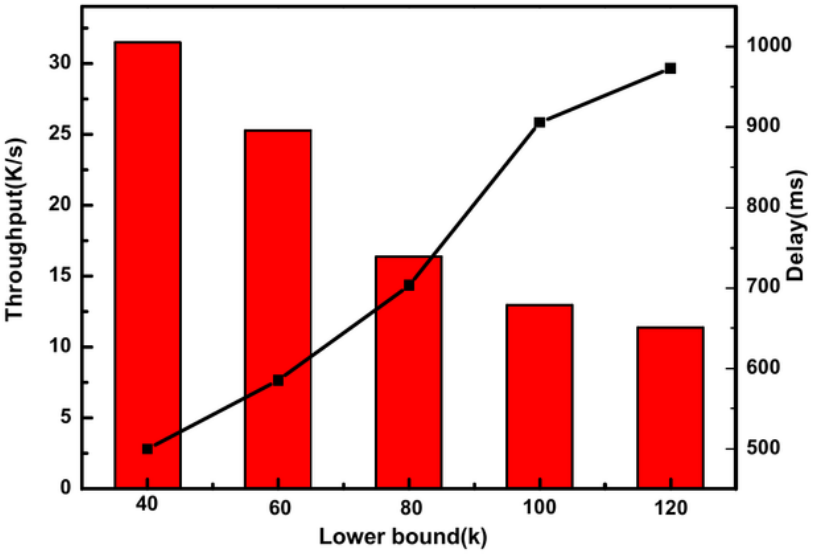


Figure 17.
Index number is $1*10^6$



To solve the low update performance of spatial index, a new spatial index QBS-tree is proposed in this paper. It uses a new top-down update algorithm, which is shown in Section EXPERIMENT to significantly reduce the computational cost of inserting and deleting index items. To index a large spatial dataset, distributed processing is an inevitable technique. In

Figure 18.
Index number is $2*10^6$

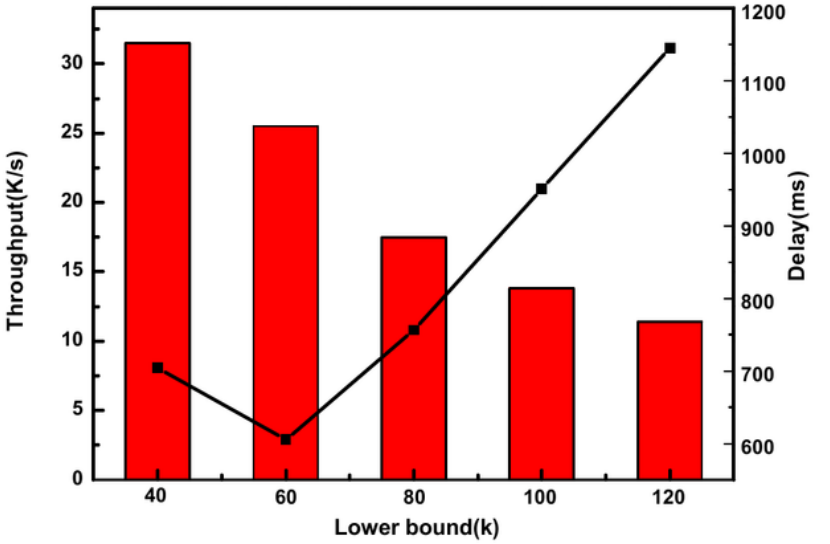
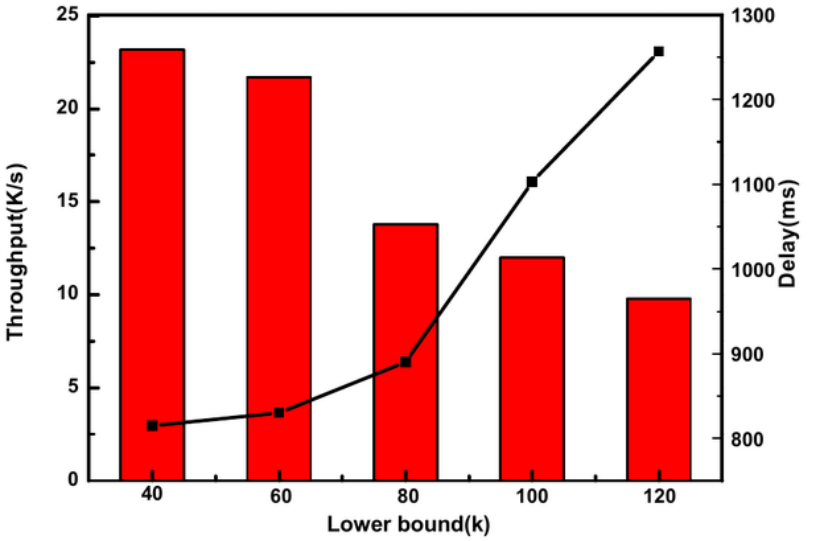


Figure 19.
Index number is $3*10^6$



addition, indexing a dynamic spatial dataset requires a dynamic load balancing technique to balance computing tasks among distributed computing nodes. In this paper, GQ-tree is used to dynamically allocate sub-index datasets among distributed computing nodes. Experimental results show that the distributed spatial index GB-QBS composed of GB-tree and QBS-tree can achieve better computational performance.

THEORETICAL AND PRACTICAL

Theoretical

QBS-tree proposed in this paper uses a new top-down update method. With this method, QBS-tree can have a configurable balance factor, making QBS-tree more tolerant of the index item's change. Therefore, the update efficiency of QBS-tree is higher than that of traditional spatial indexes. Distributed processing is inevitable for dealing with massive data. Indexing a dynamic data set in a distributed manner needs to make load balancing between parallel processing units and minimize resource consumption. The author designs a new distributed spatial index GQ-QBS with a two-layer master-slave mode to solve those problems. The global index (GQ-tree) of GQ-QBS is crucial. GQ-tree uses a density grid and a new update algorithm to balance local indexes and minimize processing units.

Practical

Real-time processing of data can often make data more valuable. Some applications, such as real-time traffic speed estimation (Nie et al., 2022), real-time spatial temporal transformer (Geng et al., 2022), and real-time crowdsourcing service (Li et al., 2021), provide more valuable real-time services by processing data in real time. These applications often require the service of spatial query, that is, to obtain objects within a specified spatial range. These index objects are changing in real time. Requires a lot of update operations on the index. The high update performance of GQ-QBS just meets this performance requirement. Through the stress test experiment in the previous section, it can be seen that GB-QBS can consume less computing resources than the existing spatial index.

ACKNOWLEDGMENT

This research was supported by National Natural Science Foundation of China under grant [No.61802273]; Postdoctoral Science Foundation of China [No.2020M681529]; and Natural Science Foundation for Colleges and Universities in Jiangsu Province [No.18KJB520044]. Finally, thanks for the data source: Didi Chuxing.

REFERENCES

Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., & Saltz, J. H. (2013). Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, *6*(11), 1009–1020. doi:10.14778/2536222.2536227 PMID:24187650

Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernandez-Moctezuma, R., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., & Whittle, S. (2015). The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, *8*(12), 1792–1803. doi:10.14778/2824032.2824076

Amaral, V. d., Giraldi, G. A., & Thomaz, C. E. (2016). A Statistical Quadtree Decomposition to Improve Face Analysis. In ICPRAM (pp. 375-380). SciTePress. doi:10.5220/0005823903750380

Beckmann, N., Kriegel, H.-P., Schneider, R., & Seeger, B. (1990). *The R*-tree: An efficient and robust access method for points and rectangles. In SIGMOD*. ACM Press.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache Flink™: Stream and Batch Processing in a Single Engine. *A Quarterly Bulletin of the Computer Society of the IEEE Technical Committee on Data Engineering*, *38*(4), 28–38.

Chen, L., Shang, S., Yao, B., & Zheng, K. (2019). Spatio-temporal top-k term search over sliding window. *World Wide Web (Bussum)*, *22*(5), 1953–1970. doi:10.1007/s11280-018-0606-x

Choy, M., Kwan, M. P., & Leong, H. V. (2000). Distributed database design for mobile geographical applications. *Journal of Database Management*, *11*(1), 3–15. doi:10.4018/jdm.2000010101

Ciaccia, P., Patella, M., & Zezula, P. (1997). *M-tree: An efficient access method for similarity search in metric spaces*. Morgan Kaufmann.

Deng, Z., Wu, X., Wang, L., Chen, X., Ranjan, R., Zomaya, A. Y., & Chen, D. (2015). Parallel Processing of Dynamic Continuous Queries over Streaming Data Flows. *IEEE Transactions on Parallel and Distributed Systems*, *26*(3), 834–846. doi:10.1109/TPDS.2014.2311811

Eldawy, A., & Mokbel, M. F. (2013). A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, *6*(12), 1230–1233. doi:10.14778/2536274.2536283

Fang, J., Zhao, P., Liu, A., Li, Z., & Zhao, L. (2019). Scalable and Adaptive Joins for Trajectory Data in Distributed Stream System. *Journal of Computer Science and Technology*, *34*(4), 747–761. doi:10.1007/s11390-019-1940-x

Fu, A. W.-C., Chan, P. M., Cheung, Y.-L., & Moon, Y. S. (2000). Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *The VLDB Journal*, *9*(2), 154–173. doi:10.1007/PL00010672

Fu, Y., Hu, Z., Guo, W., & Zhou, D. (2003). QR-tree: A hybrid spatial index structure. In *International Conference on Machine Learning and Cybernetics* (pp. 459-463). IEEE.

Geng, Z., Liang, L., Ding, T., & Zharkov, I. (2022). RSTT: Real-time Spatial Temporal Transformer for Space-Time Video Super-Resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 17441-17451). doi:10.1109/CVPR52688.2022.01692

Guttman, A. (1984). *R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD*. ACM Press. doi:10.1145/602259.602266

Hou, W. C., Yu, X., Wang, C. F., Luo, C., & Wainer, M. (2009). A dynamic grid file for high-dimensional data cube storage and range-sum querying. *Journal of Database Management*, *20*(4), 54–71. doi:10.4018/jdm.2009062503

Huang, S., Wang, B., Zhu, J., Wang, G., & Yu, G. (2014). *R-HBase: A Multi-dimensional Indexing Framework for Cloud Computing Environment. In ICDM Workshops*. IEEE Computer Society.

Jo, B., & Jung, S. (2018). Quadrant-Based Minimum Bounding Rectangle-Tree Indexing Method for Similarity Queries over Big Spatial Data in HBase. *Sensors (Basel)*, *18*(9), 3032. doi:10.3390/s18093032 PMID:30201942

Kamel, I. (1994). *Hilbert R-tree: An improved R-tree using fractals*. Morgan Kaufmann.

Kawamata, K., & Oku, K. (2019). Roadscape-based Route Recommender System Using Coarse-to-fine Route Search. *Journal of Information Processing*, *27*(0), 392–403. doi:10.2197/ipsjjip.27.392

Leutenegger, S. T., Edgington, J. M., & Lopez, M. A. (1997). *STR: A simple and efficient algorithm for R-tree packing*. IEEE Computer Society.

Li, L., Wang, L. L., & Lv, W. F. (2021). Real-time bottleneck matching in spatial crowdsourcing. *Science China. Information Sciences*, *64*(2).

Liu, Z., Chen, L., Yang, A., Ma, M., & Cao, J. (2021). HiIndex: An Efficient Spatial Index for Rapid Visualization of Large-Scale Geographic Vector Data. *ISPRS International Journal of Geo-Information*, *10*(10), 647. doi:10.3390/ijgi10100647

Lu, E. H.-C., Tseng, V. S., & Yu, P. S. (2011). Mining cluster-based temporal mobile sequential patterns in location-based service environments. *IEEE Transactions on Knowledge and Data Engineering*, *23*(6), 914–927. doi:10.1109/TKDE.2010.155

Lu, W., Shen, Y., Chen, S., & Ooi, B. C. (2012). Efficient Processing of k Nearest Neighbor Joins using MapReduce. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, *5*(10), 1016–1027. doi:10.14778/2336664.2336674

Nie, X., Peng, J., Wu, Y., Gupta, B. B., & Abd El-Latif, A. A. (2022). Real-Time Traffic Speed Estimation for Smart Cities with Spatial Temporal Data: A Gated Graph Attention Network Approach. *Big Data Research*, *28*, 100313. doi:10.1016/j.bdr.2022.100313

Nishimura, S., Das, S., Agrawal, D., & Abbadi, A. E. (2013). MD-HBase: Design and implementation of an elastic data infrastructure for cloud-scale location services. *Distributed and Parallel Databases*, *31*(2), 289–319. doi:10.1007/s10619-012-7109-z

Orlandic, R., & Yu, B. (2004). Scalable QSF-Trees: Retrieving regional objects in high-dimensional spaces. *Journal of Database Management*, *15*(3), 45–59. doi:10.4018/jdm.2004070103

Park, M.-H., Hong, J.-H., & Cho, S.-B. (2007). *Location-based recommendation system using bayesian user's preference model in mobile devices*. Springer.

Phan, T.-K., Jung, H., Youn, H. Y., & Kim, U.-M. (2017). QR*-Tree: An Adaptive Space-Partitioning Index for Monitoring Moving Objects. *Journal of Information Science and Engineering*, *33*(2), 385–411.

Shang, S., Chen, L., Wei, Z., Jensen, C. S., Zheng, K., & Kalnis, P. (2017). Trajectory similarity join in spatial networks. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, *10*(11), 1178–1189. doi:10.14778/3137628.3137630

Shang, Z., Li, G., & Bao, Z. (2018). *Dita: Distributed in-memory trajectory analytics. In SIGMOD*. ACM.

Sidlauskas, D., Ross, K. A., Jensen, C. S., & Saltenis, S. (2011). *Thread-level parallel indexing of update intensive moving-object workloads. In SSTD*. Springer.

Song, S. I., & Yoo, J. S. (2004). An efficient concurrency control algorithm for high-dimensional index structures. *Journal of Database Management*, *15*(3), 60–72. doi:10.4018/jdm.2004070104

Usman, M., Asghar, M. R., Ansari, S. I., Granelli, F., & Qaraqe, K. A. (2018). Technologies and Solutions for Location-Based Services in Smart Cities: Past, Present, and Future. *IEEE Access: Practical Innovations, Open Solutions*, *6*, 22240–22248. doi:10.1109/ACCESS.2018.2826041

Win, M. Z., Conti, A., Mazuelas, S., Shen, Y., Gifford, W. M., Dardari, D., & Chiani, M. (2011). Network localization and navigation via cooperation. *IEEE Communications Magazine*, *49*(5), 56–62. doi:10.1109/MCOM.2011.5762798

Xie, D., Li, F., & Phillips, J. M. (2017). Distributed trajectory similarity search. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, *10*(11), 1478–1489. doi:10.14778/3137628.3137655

Xiong, X., & Aref, W. G. (2006). *R-trees with update memos. In ICDE*. IEEE Computer Society.

Xu, F., Li, H., Pun, C.-M., Hu, H., Li, Y., Song, Y., & Gao, H. (2020). A new global best guided artificial bee colony algorithm with application in robot path planning. *Applied Soft Computing*, *88*, 106037. doi:10.1016/j.asoc.2019.106037

Ye, M., Yin, P., & Lee, W.-C. (2010). *Location recommendation for location-based social networks. In SIGSPATIAL*. ACM.

Yu, J., Wei, Y., Chu, Q., & Wu, L. (2021). QRB-tree Indexing: Optimized Spatial Index Expanding upon the QR-tree Index. *ISPRS International Journal of Geo-Information*, *10*(11), 727. doi:10.3390/ijgi10110727

Zaschke, T., Zimmerli, C., & Norrie, M. C. (2014). *The PH-tree: A space-efficient storage structure and multi-dimensional index. In SIGMOD*. ACM. doi:10.1145/2588555.2588564

Zhang, F., Zheng, Y., Xu, D., Du, Z., Wang, Y., Liu, R., & Ye, X. (2016). Real-time spatial queries for moving objects using storm topology. *ISPRS International Journal of Geo-Information*, *5*(10), 178. doi:10.3390/ijgi5100178

Zhang, S., Han, J., Liu, Z., Wang, K., & Xu, Z. (2009). SJMR: Parallelizing spatial join with MapReduce on clusters. In *IEEE International Conference on Cluster Computing* (pp. 1-8). IEEE Computer Society. doi:10.1109/CLUSTR.2009.5289178

Zhang, S., Ray, S., Lu, R., & Zheng, Y. (2021, August). SPRIG: A Learned Spatial Index for Range and kNN Queries. In *17th International Symposium on Spatial and Temporal Databases* (pp. 96-105). doi:10.1145/3469830.3470892

Zhang, Z., Mao, J., Jin, C., & Zhou, A. (2018). *MDTK: Bandwidth-Saving Framework for Distributed Top-k Similar Trajectory Query. In DASFAA*. Springer.

Zhou, K., Hou, Q., Wang, R., & Guo, B. (2008). Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics*, *27*(5), 1–11. doi:10.1145/1409060.1409079

Zhu, H., Liu, F., & Li, H. (2016). Efficient and Privacy-Preserving Polygons Spatial Query Framework for Location-Based Services. *IEEE Internet of Things Journal*, *4*(2), 536–545. doi:10.1109/JIOT.2016.2553083

Junhua Fang is an Associate Professor in Advanced Data Analytics Group at School of Computer Science and Technology, Soochow University, China, working with Prof. Xiaofang Zhou since August 2017. Before joining Soochow University, he earned his Ph.D degree in computer science from East China Normal University, Shanghai, in 2017, under the supervision of Prof. Aoying Zhou. He is a member of CCF. His research interests include distributed stream processing, cloud computing, distributed database, and spatio-temporal database. He has published papers in many academic conferences and journals, covering distributed stream computing, spatial index, dynamic load balancing, distributed trajectory data jion, efficient distributed stream connection, etc.

Zonglei Zhang received his bachelor's degree from Harbin Normal University, China, in 2017. At the undergraduate level, he participated in Contemporary Undergraduate Mathematical Contest in Modeling and won the provincial first prize. After graduating from university, he engaged in data warehouse related work in the insurance industry for one year. In 2018, he started to study for a master's degree at Soochow University, Chian. His research directions include spatio-temporal data, dynamic spatial index, distributed real-time computing, etc. At the postgraduate level, he was the technical leader of the power grid big data streaming computing research project, which is a cooperative project between NARI Group and Soochow University and solved the problem of insufficient scalable capabilities for power equipment monitoring tasks.