CAN-QTree: A Distributed Spatial Index for Peer-to-Peer Networks

Zuo Huaiyu, Jing Ning, Deng Yadan, Chen Luo
National University of Defence Technology
College of Electronic Science and Engineering
Changsha, China
zuohuaiyu@nudt.edu.cn

Abstract

We present a distributed spatial index called CAN-QTree based on a Content-Addressable Network (CAN) and QuadTree-alike structures. In our system, both spatial objects and their corresponding nodes in a QuadTree are identified by some points and mapped into CAN zones of peers. For a given spatial range query, CAN-QTree can provide $O(n^{1/2})$ search performance. With a uniform distribution of spatial objects, given $\varepsilon > 0$, CAN-QTree can provably maintain a load imbalance of at most $2 + \varepsilon$ between a highest loaded peer and a lightest loaded peer. Simulations show that our CAN-QTree is an effective spatial index with a guarantee of good load-balancing.

1. Introduction

As a new paradigm for structuring large-scale distributed system, peer-to-peer (p2p) systems have good scalability, resource-sharing, fault-tolerance, symmetrical nature, robustness and self-organization [1]. In current work, people often develop key-based precise queries over p2p systems. As more and more data-sensitive applications emerging, one of requirements for a p2p system is to support complex range query. Gao et al. introduced the Range Search Tree (RST) to map data partitions into nodes [2]. Range queries are decomposed into sub-queries corresponding to nodes in the RST trees. Hellerstein et al. used prefix Hilbert Space-Filling Curve in the index space to support partial keywords and range queries in p2p systems [3]. However, these works are focused on one dimensional data, thus multi-dimensional data such as spatial data can not be supported directly by those methods.

The implementation of range queries for spatial data on p2p networks has rapidly emerged as an area of interest. Grainiceanu et al. introduced a P-tree to index two-dimensional spatial objects which is very similar to the B+ tree concept but P-tree is decentralized structures [4]. However the P-tree can not support very large-scale situations for spatial data storing. Wang et al. proposed an ASPEN system based on the CAN which mapped spatial objects into some Scatter Regions [5]. Each Scatter Region contains many CAN zones where spatial objects are hashed into those zones. In the range queries, the Scatter Regions can be used as indices. But they do not analyze the quantitative load-balancing. Tanin et al. implemented a distributed spatial index based on QuadTree over a Chord p2p system [6]. They hashed the QuadTree nodes and their spatial data into the Chord p2p system and using the distributed QuadTree to support range queries. All above approaches do not balance the number of data items per peer dynamically. In this paper, we present a new distributed spatial index aiming at supporting range queries on multi-dimensional spatial data over p2p networks. Our contribution is a scheme that provably maintains a load imbalance of at most $2 + \varepsilon$ (for given $\varepsilon > 0$) while achieving amortized constant cost per insertion and deletion. The CAN-QTree also provides search performance of $O(n^{1/2})$.

The rest of this paper can be organized as follows. Section 2 gives some backgrounds. Section 3 presents management of peers in CAN-QTree and maintenance of CAN-QTree. Section 4 reports simulation results using our index. And Section 5 contains concluding remarks.

2. Background

2.1. Content-Addressable Network

Content-Addressable Network (CAN) [8] manages data storing and queries based on a virtual d-dimensional Descartes coordinate space. A CAN is similar to a very large hash table which is composed of many peers, in which every peer maintains a part of the CAN space called a zone. And each zone has a Virtual Identifier (VID). The whole virtual Descartes coordinate space is decomposed to all peers dynamically, so each peer has a self-governed zone. During key-based item storing, namely *(key,value)= (K,V)*, the

978-0-7695-3352-0/08 $25.00 © 2008 IEEE
DOI 10.1109/HPCC.2008.20

key K will be hashed into a zone where a corresponding peer will accept the item. When querying the item, every peer can use a uniform hash function to find K and its corresponding peer. Fig.1 shows a virtual 2-dimensional Descartes coordinate space divided into six zones and its Virtual Identifier tree.

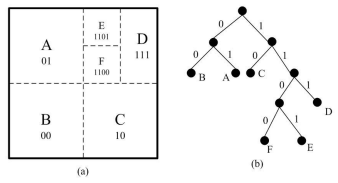


Figure 1. Zones of CAN and its VID tree

2.2. QuadTree Index

Spatial range query can be divided into two phases. The first phase is approximate match based on some index structures and the second phase is precise match. In a QuadTree-like data structures, spatial objects are approached by some little squares called tiles which are decomposed from the whole underlying space. There are many variants of the QuadTree data structures, with the region QuadTree being the most common. In this case, the underlying two-dimensional tile-shaped space is recursively decomposed into four congruent tiles until each tile is not contained in any of the spatial object. We choose MX-CIF QuadTree [8] to exhibit our P2P index and associated algorithms although other QuadTree types could have also been utilized. Fig.2 depicts some spatial objects indexed by our QuadTree.

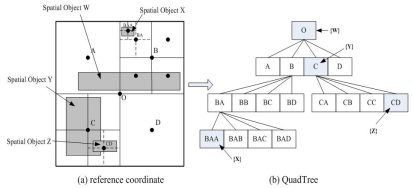


Figure 2. An example for QuadTree index

Each node in a QuadTree can be uniquely identified by its centroid, named a control point. For example, the black points O,A,B,BA are control points shown in Fig.2 The control point can be computed using the globally known QuadTree subdivision method to recursively subdivide the underlying space. Each control point u has the following data structure associated with it: $D(u) = \{d_1, d_2, d_3, d_4, list\}$. $d_i \in \mathbb{N}$ are downward counts used to indicate the number of spatial objects which exist at or below child i. list is a list of spatial objects that intersect the region *R(u)* and that could not be inserted at a deeper level in the QuadTree. Throughout the paper, *R(u)* denotes the region defined by a QuadTree tile centered at control point u. Also we use *L(u)* to denote the level or depth of the tree at which control point u is present, and *C(u, i)* to represent the i th child of control point u, where i=1,2,3,4. When a spatial object is indexed by the QuadTree, it also has a similar control point as a tree node does.

3. CAN-QTree

If we can attach a peer to a node of the QuadTree as well as a zone of the CAN space, then that peer is responsible for all queries that intersect that node, and for storing the spatial objects that are associated with that node region. Because each QuadTree node or spatial object can be identified by a control point, we can use the CAN method to hash those control points so that the responsibility for a node region is associated with a peer.

3.1. Index Maintenance

In this section, we present the details of the operations for our CAN-QTree. If a QuadTree has centralized structure, those procedures are usually all "one-pass" algorithms that proceed downward from the root of the tree, branching decision at each node. Since our operations are decentralized, the top-down maintenance method will cost much more for sending spatial objects from a top-level to its subtree. And also, the top-down method will make the root of the tree very busy. Hence, we choose another method, namely, bottom-up method, to maintain a CAN-QTree index.

3.2.1. Insertion

Algorithm 1: InsertObject(object X)
1. control point u=Subdivide(X, root);
2. $Delegate(u) \rightarrow D(u).list$ +=X;
3. control point u'=u.parent;
4. while (u' is not empty)
5. $\quad Delegate(u') \rightarrow D(u').d_i(u)$ +=1;
6. $\quad u'$=u'.parent

Function 1: Subdivide(object X, control point u)
1. if (X not within one $R(C(u,i))$ or $(L(u) = f_{\max})$)
2. $\quad$ return u;
3. else
4. $\quad$ for i=1 to 4 do
5. if(Ints(X, $R(C(u,i))$) is not empty) then
6. $\quad$ Subdivide(X,$C(u,i)$);

We use *Delegate* to mean that a peer sends a message that invokes a function on another peer. A peer calls *InsertObject* to insert an object into the tree. Algorithm 1 first computes a spatial object X's control point u using function *Subdivide*, and then routes it to another peer to store

the object X and its control point u. After that, algorithm 1 recursively updates the whole tree.

The messages are produced by step 2, step 4 and step 5. According to the route protocols of a CAN [7], the step 2 will create $O((d/4) \cdot n^{1/d})$ messages and step 4 and step 5 will bring $O((d/4) \cdot n^{1/d} \cdot f_{\max})$ messages with a given $f_{\max}$ levels QuadTree. Hence, the total messages will be $O((d/4) \cdot n^{1/d} \cdot (f_{\max}+1)) \approx O(n^{1/2})$. We also can see that a message produced by step 5 only contains a small data (d_i) rather than a spatial object data. Since each node of a distributed CAN-QTree has a fixed parent, we can allow each node to maintain a cache of addresses for its parent and thereby reduce the delegation message complexity to $O(1)$.

3.2.2. Deletion. We use the following algorithm to delete a spatial object from a CAN-QTree.

Algorithm 2: DeleteObject(object X)
1. control point u=Subdivide(X, root);
2. $Delegate(u) \rightarrow D(u).list$ =X;
3. control point u'=u.parent;
4. while (u' is not empty)
5. $Delegate(u') \rightarrow D(u').d_i(u)$ =1;

Algorithm 2 is almost identical to algorithm 1 and thus is not analyzed here. The main different is that the spatial object is removed from $D(u).list$ and that $D(u).d_i$ is decremented by 1.

3.2.3. Query. Algorithm 3 presents a spatial range query algorithm with a CAN-QTree tree index. When a peer in the system receives a spatial range query request, the peer first calculates how many tree nodes identified by control points overlap with the query window on line 1. And then the peer forwards the query message to other peers with those control points recursively on line 2 and line 3.

Algorithm 3: RangeQuery (query Q)
1. control point list G=DecmpQry(Q, root);
2. for each u in G do in parallel
3. $Delegate(u)$→DoQuery(Q);

Function 2: DecmpQry(query Q, control point u)
1. control point list G={};
2. if (Ints(Q, $R(u)$) is not empty)then
3. G+=u;
4. for i=1 to 4
5. if(Ints(Q,$R(C(u,i))$ not empty) and $D(u).d_i(u)$>0) then
6. G+=u;
7. G+=DecmpQry(Q, $C(u,i)$);
8. return G;

We assume the d dimensional space partitioned into n equal zones, individual peers maintain $2d$ neighbours, and the average routing path length is still $O((d/4) \cdot n^{1/d}) \approx O(n^{1/2})$ [7]. Hence line 3 will produce $O(n^{1/2})$ messages. Let we assume that zones are uniformly distributed in the CAN-QTree space. Let c denote the size of each zone. Let q denote the length of each side of a spatial range query Q. Then q can be represented as $q = i \times c + x$, where $x \subset [0, c)$ and $i \in \mathbb{N}$. It can be verified that q will cover $[(i+1)^2, (i+2)^2]$ zones. So i will be $(q+c)^2/c^2$ on the average. In the worst case where query Q covers the complete space of CAN-QTree, query q will cover $(q+c)^2/c^2 = n$ zones. For each peer, its parent will produces messages, too. So the number of peers involved will be $O(n)$+ $O(log_4^n)$+ $O(\log_4^{\log_4^n})$+...≈ $O(n)$. Then the message count complexity will be $O(n^{3/2})$ in algorithm 2. However, for a given query q, the message count complexity will be $O(n^{1/2})$ with a constant number of peers involved.

3.2. Load Balancing

Techniques developed for equality queries have good load-balance as they distribute original data items based on their hash values. Since hashing destroys the order of data items, spatial range queries can not be answered efficiently. In our CAN-QTree system, we first identify a spatial object by its control point u, then use u as a key value to hash its spatial object onto a corresponding peer. In such approach, if the control points have skewed distributions, the system will still get a bad load-balance because many spatial objects will have a same control point. This case often happens within a real spatial dataset.

A simple approach to maintain the load-balance of a peer is to record the owner peer with the number of its data items and dynamic redistribute those data items according to the number. Whenever the number of data items in a peer becomes larger than a given parameter due to many insertions, the peer will try to split its assigned data items with other peers. In above approach, we are faced with two challenges. First, during splitting or emerging, how many cost do we pay for those rebalance operations? And second, how do we balance the loads among those peers fairly?

In order to decrease the cost of re-balancing operations, we use collaborative peers ($CPeers$) for re-balance operations. A $CPeer$ is a special kind of peer in our CAN-QTree which is not assigned any zone as a CAN peer does. We denote the set of collaborative peers as C and the set of normal CAN peers ($NPeers$) as N. We assume individual normal peers associate $p.own$ to denote the list of all spatial objects assigned by a base hash function. And let $[\mu_{\min}, \mu_{\max}]$ denotes the load bounds assigned to each peer. Whenever the number of spatial objects in a NPeer p satisfies $|p.own| \geq \mu_{\max}$ (or $|p.own| < \mu_{\min}$), it will try to split (or emerge) its data items with a CPeer.

In order to keep the symmetry of a CAN system, we use very small number of CPeers and we can see from the latter load-balance algorithm that the number of CPeers will change over time during re-balancing. To find a CPeer, we create an artificial item $(u_0, p_c.addr)$ for every CPeer p_c, where u_0 is a control point of a deepest node in the CAN-

QTree tree. This item is inserted into the system like any spatial object. For a given CAN-QTree with m levels, the cost of finding a CPeer is $O(n^{1/2}/2)$.

3.3.1. Load-balance Algorithm. Now we discuss how to utilize the CPeers. Suppose a NPeer has k CPeers $\{q_1, ..., q_k\}$. For individual CPeers q_i, we look for a peer from the set of $p \cup \{q_1, q_2, ..., q_{i-1}, q_{i+1}, ..., q_k\}$ having a maximal zone and then divide the zone into two equal parts such that the peer manages one part and the q_i manages the other. Hence, each CPeer of p will become responsible for some spatial objects from p. Assume the NPeer has a zone with a S_p area, then after assignment its zone will be $p.zone.S_p \leq 2^{-\left\lceil \log_4^{(k+1)} \right\rceil} \cdot S_p$. Therefore the NPeer will not be responsible for all its original spatial objects and be able to maintain the assigned bounds of its load. In this context, we assume $p.own$ denote the list of all these spatial objects and $p.resp$ denote the list of the spatial objects which actually are stored on it.

The split algorithm is as following:

Algorithm 4: split(NPeer p)
1. if ($p.CPeers.count > 0$) then
2. p_c=getFirstCPeer();
3. else
4. p_c=searchCPeer();
5. if (p_c is null) then
6. return;
7. else
8. $p_c.own$=$p.own$.splitItems();
9. p_c.scatterZone=p.scatterZone=p.zone;
10. p_c.zone=p.zone.splitZone();
11. p_c.joinZone();
12. redistribute $p.own$ with its reduced set of CPeers;

Algorithm 4 shows the pseudo-code of the split operation executed by a NPeer p. The splitting peer p either gets a CPeer p_c from its list of CPeers or searches for a new CPeer p_c from the system on line 2 and line 4. Line 8 transfers half of the NPeer items to the p_c. Line 9 assigns a scatter zone to the NPeer p and the CPeer p_c. Line 10 transfers half of the corresponding zone to the p_c and line 11 let the p_c joins our system. At last, the NPeer p redistributes its spatial objects with its reduced set of CPeers. In algorithm 4, we use a scatter zone to redistribute a highly loaded peer from line 8 to line 10. Here, we give the formally concept of a scatter zone.

Assume the logical space used by CAN-QTree is denoted as a set of zones $Z = \{z_1, ..., z_n\}$ and spatial objects are mapped into this space according to their control points. A $scatter\ zone$ is a composite zone which satisfies $SZ = \cup_{i=1}^{k} z_i (z_i \in Z)$, where k is a scatter factor.

A scatter zone contains several normal zones and their associated NPeers. Spatial objects located in a specific scatter zone are uniformly distributed across the zones within a scatter zone. For a given spatial object X and its $MBR(x_1, y_1, x_2, y_2)$, we hash the central point of MBR $u_{mbr} = ((x_1 + x_2)/2, (y_1 + y_2)/2)$ into those zones. The purpose of scatter zones is to constrain the distribution of spatial objects and uniformly scatter them within a scatter zone.

The scatter factor k is to control a scatter zone's size. Larger sized scatter zones result in a more uniform distribution of spatial objects. However, they also result in a larger search area for spatial range queries. We will plot an optimal value of k in the stimulation experiments.

The merge algorithm is as following:

Algorithm 5: merge(NPeer p)
1. p'=getNeighborForMerge();
2. if (p' is null) then
3. usurp(p);
4. else
5. $Delegate(p')$ →addObjects($p.own$);
6. p.leaveZone();
7. $Delegate(p')$ → addCPeer(p);

Algorithm 6: usurp(NPeer p)
1. (q,p')=getLeastLoadedCPeer();
2. if $|p.resp| \geq \mu_{\max}\sqrt{1+\omega}|q.resp|/\mu_{\min}$ then
3. p.setCPeer(q);
4. redistribute $p.own$ among new set of CPeers;
5. redistribute $p'.own$ among new set of CPeers;

In algorithm 5, a NPeer p first looks for a neighbour p' with a merging desire. Line 3 calls $usurp$ to take over a CPeer q of another NPeer for redistributing if there is no such neighbour peer. Line 5 to line 7 moves spatial objects from p to p' and lets p become one of $p's$ CPeers.

Algorithm 6 introduces an additional load balancing operation called $usurp$. During usurp, a NPeer p can usurp a CPeer q from another NPeer p' if $|p.resp| \geq \mu_{\max}\sqrt{1+\omega}|q.resp|/\mu_{\min}$. With the help of q, the $p.resp$ will be reduced.

Query algorithm does not change much if we use above load-balance algorithms. When a NPeer executes $DoQuery$, it first checks whether it belongs to a scatter zone. With a yes, the NPeer floods the query inside the scatter zone.

3.3.2. Load-balance Analysis. We are analyzing our load-balance strategy introduced above. We assume the set of NPeers to be N, the set of CPeers to be C and the set of all peers to be W. We also define a load imbalance as $\delta = (\max_{p \in N} |p.resp|)/(\min_{p \in N} |p.resp|)$. And we use amortized analysis [9] method to discuss our load-balance.

THEOREM 1: For every sequence of spatial objects insertions or deletions, and constants $\mu_{\max}$,$\mu_{\min}$ and $\varepsilon > 0$ such that $\mu_{\max}/\mu_{\min} \geq 2$, if central points of spatial objects' MBR in a data set have a uniformly distribution, the system satisfies those following constrains:

(1) Load imbalance $\delta < \max(2 + \varepsilon, \mu_{\max}/\mu_{\min})$

(2) If $\mu_{\max}/\mu_{\min} \geq 2 + \varepsilon$, the $split$, $merge$ and $usurp$

operations is such that the amortized cost of an insertion or a deletion operation is a constant.

Proof:We first proof the conclusion 1. It is easy to see that during our load-balance course, the split and the merge operations bound the size of $p.own$ within $\mu_{\max}$ and $\mu_{\min}$. If a NPeer p has some CPeers, then it could have CPeers making $|p.resp| < \mu_{\min}$. For a NPeer p without CPeers, it can usurp a CPeer q from another NPeer p' where exists $|p.resp| \geq \mu_{\max}\sqrt{1+\omega}|q.resp|/\mu_{\min}$. By setting $\omega = (1+\varepsilon/2)^2 - 1$, we can get the conclusion 1.

We now give the proof for the second conclusion. Let $\mu_{\max}/\mu_{\min} = 2+\varepsilon$ and the potential function $\Phi = \Phi_{own} + \Phi_{resp}$, where

(1)$\Phi_{own} = \sum_{p\in W} \phi_{own}(p)$

(2)$\Phi_{resp} = \sum_{q\in W} \phi_{resp}(q)$

(3)$\phi_{own}(p) = \begin{cases} 0 & p \in C \\ X & \mu_{\min} \leq |p.own| < \mu^0_{\min} \\ 0 & \mu^0_{\min} \leq |p.own| < \mu^0_{\max} \\ Y & \mu^0_{\max} \leq |p.own| \leq \mu_{\max} \end{cases}$

where

$\mu^0_{\min} = (1+\frac{\varepsilon}{4})\mu_{\min}$ and $\mu^0_{\max} = (2+\frac{3\varepsilon}{4})\mu_{\min}$

$X = \frac{c_{own}}{\mu_{\min}}(\mu^0_{\min} - |p.own|)^2$

$Y = \frac{c_{own}}{\mu_{\min}}(|p.own| - \mu^0_{\max})^2$

(4)$\phi_{resp} = \frac{c_{resp}}{\mu_{\min}}(|q.resp|)^2$

Before proving the second conclusion of theorem 1, we introduce the following three lemmas.

Lemma 1: Assume system parameters have the same values as in the Theorem 1, the cost of split algorithm will be lower than the minimum decrease in the potential Φ due to split with the condition $c_{own} \geq 24(2+\varepsilon)/\varepsilon^2$.

Proof: We firstly compute $\Delta_{split}\Phi_{own}$ as the following steps:

A splitting NPeer p has $|p.own| = (2+\varepsilon)\mu_{\min}$, after redistribution p and q will belong to a same scatter zone and be responsible for respective $(1+\varepsilon/2)\mu_{\min}$ spatial objects. According to the definitions of the function Φ, before the split we have

$\phi_{own}(q) = 0$ and $\phi_{own}(p) = \frac{c_{own}}{\mu_{\min}}(|p.own| - \mu^0_{\max})^2$

After the split we have

$\phi_{own}(q) = 0$ and $\phi_{own}(p) = 0$

So we can obtain

$$\begin{aligned}\Delta_{split}\Phi_{own} &= -\tfrac{c_{own}}{\mu_{\min}}(|p.own| - \mu^0_{\max})^2 \\ &\leq -c_{own}\cdot(\tfrac{\varepsilon}{4})^2\mu_{\min}\end{aligned}$$

We secondly calculate $\Delta_{split}\Phi_{resp}$ as the following cases:

(1) Suppose $p.CPeers$ is not empty and let it be $\{q_1, q_2, ...k\}$. Before the split we have

$\phi_{resp}(q_i) = \frac{c_{resp}}{\mu_{\min}}(\frac{(2+\varepsilon)\mu_{\min}}{k})^2$

After the split, there exists two NPeers p and q where

$p.CPeers$=$\{q_1, q_2, ..., q_m\}(m = \lceil\frac{k-1}{2}\rceil)$

$q.CPeers = \{q_{m+1}, ..., q_{m+n}\}(n = \lfloor\frac{k-1}{2}\rfloor)$

So we can get

$\phi_{resp}(q_i) = \begin{cases} \frac{c_{resp}}{\mu_{\min}}(\frac{(1+\varepsilon/2)\mu_{\min}}{m})^2 & q_i \in p.CPeers \\ \frac{c_{resp}}{\mu_{\min}}(\frac{(1+\varepsilon/2)\mu_{\min}}{n})^2 & q_i \in q.CPeers \end{cases}$

Thus we have

$$\begin{aligned}\Delta_{split}\Phi_{resp} = m\cdot\tfrac{c_{resp}}{\mu_{\min}}(\tfrac{(1+\varepsilon/2)\mu_{\min}}{m})^2 \\ +n\cdot\tfrac{c_{resp}}{\mu_{\min}}(\tfrac{(1+\varepsilon/2)\mu_{\min}}{n})^2 \\ -k\cdot\tfrac{c_{resp}}{\mu_{\min}}(\tfrac{(2+\varepsilon)\mu_{\min}}{k})^2\end{aligned}$$

In the case of $m = n$, we can get $\Delta_{split}\Phi_{resp} = 0$. And in the case of $m \neq n$, we can get

$$\begin{aligned}\Delta_{split}\Phi_{resp} &\leq \tfrac{c_{resp}}{\mu_{\min}}(\tfrac{((2+\varepsilon)\mu_{\min})^2}{4m} - \tfrac{((2+\varepsilon)\mu_{\min})^2}{4n}) \\ &\leq \tfrac{c_{resp}(2+\varepsilon)^2\mu_{\min}}{8}\end{aligned}$$

(2) Assume $p.CPeers$ is empty. During the split, the NPeer p finds a CPeer q where $q \in p_0.CPeers$, so we have

$\Delta\phi_{resp}(p) = \frac{c_{resp}}{\mu_{\min}}((1+\varepsilon/2)\mu_{\min} - (2+\varepsilon)\mu_{\min})^2$

$$\begin{aligned}\Delta_{split}\phi_{resp}(q) = \tfrac{c_{resp}}{\mu_{\min}}(((1+\varepsilon/2)\mu_{\min})^2 \\ -(\tfrac{|p_0.own|}{1+p_0.CPeers.Count})^2)\end{aligned}$$

For $\forall q_0(\neq q) \in (p_0.CPeers \cup \{p_0\})$, we can get

$\Delta_{split}\phi_{resp}(q_0) = \frac{c_{resp}}{\mu_{\min}}((\frac{|p_0.own|}{p_0.CPeers.Count})^2$

$$-(\frac{|p_0.own|}{1+p_0.CPeers.Count})^2)$$

That is

$$\Delta_{split}\Phi_{resp} = \frac{c_{resp}}{\mu_{\min}}(\frac{|p_0.own|^2}{p_0.CPeers.Count} - \frac{1}{2}((2+\varepsilon)\mu_{\min})^2 - \frac{|p_0.own|^2}{1+p_0.CPeers.Count})$$

Since we have $|p_0.own| \leq (2+\varepsilon)\mu_{\min}$ and $p_0.CPeers.Count \geq 1$, we know that $\Delta_{split}\Phi_{resp} \leq 0$.

Hence, there is an equation

$$\Delta_{split}\Phi = \Delta_{split}\Phi_{own} + \Delta_{split}\Phi_{resp} \leq -c_{own} \cdot (\frac{\varepsilon}{4})^2\mu_{\min}$$

Consider the cost of split is

$$\mu^0_{\min} + \mu^0_{\max} = (1+\frac{\varepsilon}{4})\mu_{\min} + (1+\frac{3\varepsilon}{4})\mu_{\min}$$

In order to have the cost of split lower than the decrease in the potential due to split, we need

$$-c_{own} \cdot (\frac{\varepsilon}{4})^2\mu_{\min} + (1+\frac{\varepsilon}{4})\mu_{\min} + (1+\frac{3\varepsilon}{4})\mu_{\min} > 0$$

That is

$$c_{own} \geq 24(2+\varepsilon)/\varepsilon^2$$

Analogous to the lemma 1 case, we can prove the two following lemmas. Due to space limitation, we omit their proofs.

Lemma 2: Assume system parameters have the same values as in the Theorem 1, the cost of merge algorithm will be lower than the minimum decrease in the potential Φ due to split with the condition $c_{own} \geq (32c_{resp} + 48 + 16\varepsilon)/\varepsilon^2$.

Lemma 3: Assume system parameters have the same values as in the Theorem 1, the cost of usurp algorithm will be lower than the minimum decrease in the potential Φ due to split with the condition $c_{resp} \geq (2+\varepsilon)k^2/\omega$ where $k = (4+\varepsilon)((8+4\varepsilon)\sqrt{1+\omega} - 1)$.

During insert operation, a spatial object is inserted into $p.own$ or $q.resp$ according to our algorithm where $p \in N$ and $q \in p.CPeers \cup \{p\}$. In this case, the potential function $\phi_{resp}(q)$ increases, while $\phi_{own}(p)$ increases if $\mu^0_{\max} \leq |p.own| \leq \mu_{\max}$ and decreases if $\mu_{\min} \leq |p.own| \leq \mu^0_{\min}$. So the maximum increase in Φ occurs when both $\phi_{resp}(q)$ and $\phi_{own}(p)$ increase and this increase is

$$\Delta_{insert}\Phi_{own} = \Phi_{own}(i) - \Phi_{own}(i-1)$$

$$= \frac{c_{own}}{\mu_{\min}}(|p.own|+1-\mu^0_{\max})^2 - \frac{c_{own}}{\mu_{\min}}(|p.own|-\mu^0_{\max})^2$$

$$\leq \frac{c_{own}\cdot\varepsilon}{2} + c_{own}$$

$$\Delta_{insert}\Phi_{resp} = \Phi_{resp}(i) - \Phi_{resp}(i-1)$$

$$= \frac{c_{resp}}{\mu_{\min}}(|q.resp|+1)^2 - \frac{c_{resp}}{\mu_{\min}}(|q.resp|)^2$$

$$\leq 5c_{resp} + 2\varepsilon c_{resp}$$

According to the above equations, we get

$$\Delta_{insert}\Phi = \Delta_{insert}\Phi_{own} + \Delta_{insert}\Phi_{resp} \leq \frac{c_{own}\cdot\varepsilon}{2} + c_{own} + 5c_{resp} + 2\varepsilon c_{resp} \quad (1)$$

Analogous to the insertion case, we can prove that during a deletion the maximum increase in potential to be

$$\Delta_{delete}\Phi = \Delta_{delete}\Phi_{own} + \Delta_{delete}\Phi_{resp} \leq \frac{c_{own}\cdot\varepsilon}{2} + c_{own} + 5c_{resp} + 2\varepsilon c_{resp} \quad (2)$$

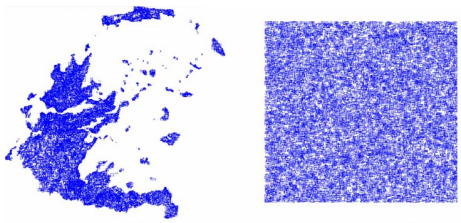
According to equations (1) and (2) we can prove that the increase of potential function Φ is a constant during a sequence of insertions and deletions. Due to lemma 1, lemma 2 and lemma 3, we know the cost of the split, merge and usurp are all lower than the decrease in the potential due to the corresponding operations by setting the appropriate constants c_{own} and c_{resp}. So the amortized constant cost for an insertion or a deletion is a constant.

Theorem 1 shows that the amortized cost of an insertion or a deletion is $O(1)$. For a sequence of n insertions and deletions, the amortized cost will be $O(n)$, thus the actual cost of n insertions and deletions is $O(n)$ at the worst case. Theorem 1 guarantees that our CAN-QTree can provide a good load-balance algorithm with a maintenance cost of $O(n)$.

4. Experimental Evaluation

We evaluate our system on both synthetic and real data sets. The synthetic data set (SDS) consists of 32768 rectangles where each rectangle is randomly produced which is no more than 1/212 of the whole logical CAN space, shown as Fig.3(b). The real data set (RDS) contains information about the 32678 minimal bounding boxes (MBR) of Grecian rivers shown as Fig.3(a). We developed a simulator in C++ to evaluate the index structures. The simulation was executed on a personal computer with a 1024M memory, a 3.2GHz Celeron CPU and a 160G 7200RPM SATA disk. In the experiments, we implemented three kinds of

CAN-QTree. The first is *CAN-QTree-0* which has no load-balancing. The second is *CAN-QTree-1* which has a load-balancing given by section 3.2 and the third is *CAN-QTree-2* which maintains a cache of addresses for the tree node's parent and without load-balancing. All the following experiments will associate a scatter factor $k = 2$ and there were total 1024 peers in system. We set the levels of the CAN-QTree to be 10, $\mu_{\min}$ to be 32, $\mu_{\max} = 2\mu_{\min}$ and $\varepsilon = 0.5$.



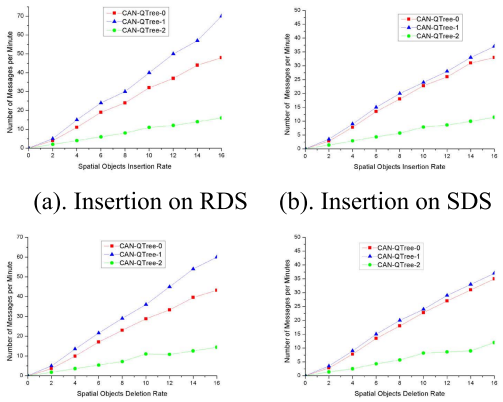
(a). RDS (b). SDS

Figure 3. The data sets of experiments

We were first interested in how the insertion or deletion operations affect the message cost required for maintaining a CAN-QTree index. Fig.4 plots the message cost as a function of the varying insertion or deletion operations rate where peers are set to be 128 and the operations rate is from 2 operations per minute to 16 operations per minute. The results show that the CAN-QTree-2 has a minimal message cost and the CAN-QTree-1 has a maximal message cost. The CAN-QTree-2 has a cache strategy which reduce the update message complexity to $O(1)$, so it has a minimal maintaining message cost. The CAN-QTree-1 needs to rebalance its spatial objects when updating, so it has a maximal maintaining message cost. The results also show that the difference between CAN-QTree-1 and CAN-QTree-0 is not so great, especially on synthetic data set shown as Fig.4(b) and Fig.4(d).

Fig.5 shows the search cost on a CAN-QTree index as a result of varying peers in system. We generated a set of rectangular query windows and each query window size is equal to 1/512 of the whole CAN space. All those query windows are randomly distributed into the CAN space. We use the average search cost among 100 queries results. In this experiment, there are 32768 spatial objects stored in the system and the number of peers varies from 128 to 1024. From Fig.5, we can see that CAN-QTree-2 still have a best performance for its cache strategy. It also can be seen that CAN-QTree-0 beats CAN-QTree-1 because CAN-QTree-1 has scatter zones which will produce more messages due to flooding queries inside each scatter zone. Fig.5(b) also shows that CAN-QTree-1 and CAN-QTree-0 almost have a same performance for the reason that the synthetic data set has a uniform data distribution which actually decrease the number of scatter zones in CAN-QTree-1.

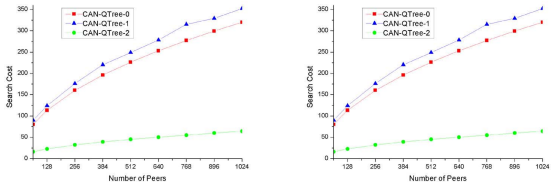
We are now exploring the load-balance of the CAN-



(a). Insertion on RDS (b). Insertion on SDS

(c). Deletion on RDS (d). Deletion on SDS

Figure 4. The message cost required for maintaining a CAN-QTree index

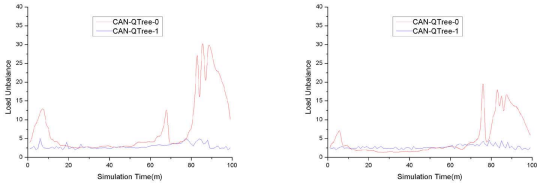


(a). RDS (b). SDS

Figure 5. Search cost on a CAN-QTree index

QTree system. We study the performance of the system as spatial objects are inserted and deleted from the system. To see the effect of insertion and deletion operations on the load balance, we start the system by initiating 64 peers and no stored spatial objects. Then, we randomly insert or delete spatial objects in three phases: insert only, insert and delete, and delete only. In each phase we carry out 2000 operations at a rate of 1 operation per second. The load-imbalance is measured per minute. Since the CAN-QTree-2 has a same performance with the CAN-QTree-0, we do not consider it any more. Fig.6 demonstrates the different performances between CAN-QTree-0 and CAN-QTree-1. From those results, we can see that CAN-QTree-0 has a limited load-balance ability which only performs well during the second phase. However, CAN-QTree-1 has a steady load-balance ability which performs well throughout the experiment. So we can conclude that CAN-QTree-0 will face more changes while CAN-QTree-1 adapts itself well in a skewed data distribution situation.

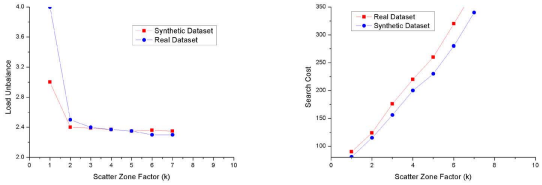
Using this experiment, we plot effects of the scatter factor on CAN-QTree system. For this experiment, there are total 1024 peers in the system hosting 32678 spatial objects from synthetic data set (the Grecian data set has the



(a). RDS (b). SDS

Figure 6. The load-balance performance of a CAN-QTree

same conclusions). The schemes of queries are same as the above one in Fig4. Fig.7 gives the load-imbalance and the search cost with a varying scatter factor k. we can make an observation from Fig.7 that the load-imbalance deceases as k increases while the search cost increases as k increases. When we analyzed the results for this experiment, we saw that there was a drastic decrease in load-imbalance when k changes from 1 to 2 while the search cost increases linearly all the while with an increased k. So this experiment tells that the scatter factor k has an appropriate optimal value which is 2.



(a). load-imbalance with k (b). search cost with k

Figure 7. CAN-QTree system performance with a varying of scatter factor k

5. Conclusions

We have introduced CAN-QTree, a novel distributed index structure that efficiently supports spatial range queries in a CAN P2P environment based on a QuadTree index structure. In our CAN-QTree system, each node of the index tree is identified by a control point. So does spatial objects. With a global hash function, we can scatter those tree nodes and spatial objects onto the CAN peers by hashing their control points. To effectively balance spatial objects among peers, we utilize collaborative peers and scatter zones to re-balance and provide provable guarantees on maintaining cost. Our experimental evaluation shows that CAN-QTree performs well.

6. Acknowledgment

This work is supported by National Science Foundation of China under Grant No. 40601080 and the National High Technology Research and Development Plan of China under Grant No. 2006AA12Z205.

References

[1] C. Wang and B. Li. Peer-to-peer overlay networks: A survey. Technical Report DEC-TR-506, Department of Computer Science, HKUST, Feb 2003.

[2] J. Gao and P. Steenkiste. An adaptive protocol for efficient support of range queries in dht-based systems. In *12th IEEE International Conference on Network Protocols*, 2004.

[3] J. Hellerstein, S. Ratnasamy, and S. Shenker. Range query over dhts. Technical Report IRB-TR-03-009, Intel Research, Berkeley, June 2003.

[4] A. Grainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *ACM SIGMOD'04 WebDB Workshop*, Paris, France, 2004.

[5] H. Wang, R. Zimmermann, and W.-S. Ku. ASPEN: An adaptive spatial peer-to-peer network. In *GIS'05*, Bremen, Germany, November 2005.

[6] E. Tanin, A. Harwood, and H. Samet. Using a distributed quadtree index in peer-to-peer networks. *VLDB Journal*, 16(2):165–178, 2007.

[7] S. Ratnasamy. A scalable content-addressable network. Technical report, Ph.D. Dissertation University of California, Berkeley, 2002.

[8] K. G. The quad-CIF tree: A data structure for hierarchical online algorithms. In *the 19th Design Automation Conference*, pages 352–357, Las Vegas, 1982.

[9] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Descrete Methods*, 6(2):306–318, 1985.