Egemen Tanin · Aaron Harwood · Hanan Samet

Using a Distributed Quadtree Index in Peer-to-Peer Networks

Abstract Peer-to-peer (P2P) networks have become a powerful means for online data exchange. Currently, users are primarily utilizing these networks to perform exact-match queries and retrieve complete files. However, future more data intensive applications, such as P2P auction networks, P2P job-search networks, P2P multi-player games, will require the capability to respond to more complex queries such as range queries involving numerous data types including those that have a spatial component. In this paper, a distributed quadtree index that adapts the MX-CIF quadtree is described that enables more powerful accesses to data in P2P networks. This index has been implemented for various prototype P2P applications and results of experiments are presented. Our index is easy to use, scalable, and exhibits good load-balancing properties. Similar indices can be constructed for various multi-dimensional data types with both spatial and non-spatial components.

This work was supported in part by the National Science Foundation under grants EIA-99-00268, EIA-00-91474, and CCF-05-15241 as well as Microsoft Research.

Egemen Tanin
NICTA Victoria Laboratory
Department of Computer Science and
Software Engineering
University of Melbourne
Victoria 3010, Australia
E-mail: egemen@cs.mu.oz.au

Aaron Harwood
NICTA Victoria Laboratory
Department of Computer Science and
Software Engineering
University of Melbourne
Victoria 3010, Australia
E-mail: aharwood@cs.mu.oz.au

Hanan Samet
Department of Computer Science
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland, College Park
Maryland 20742, USA
E-mail: hjs@cs.umd.edu

Keywords Quadtrees, Spatial Data Structures, Distributed Data Structures, Peer-to-Peer Networks

1 Introduction

Internet users are ready to adapt new Peer-to-Peer (P2P) applications to exchange a greater variety of data than just music files. Some examples of such applications include P2P auctions, P2P job-search networks, and P2P multi-player games, to name a few. These applications, without requiring a central service provider nor a single point of failure and control, can form low-cost, easy to deploy, and scalable medium of data exchanges. Accessing many resources in a coherent manner and finding data on such highly dynamic distributed environments is a challenging problem. Researchers have been working on various indices that can be used to efficiently access data on P2P networks. Examples of this work are [23, 27, 28, 35, 38]. The bottleneck for the new P2P applications is that often these indices cannot provide the necessary functionality to perform many types of complex queries on complex data that users have come to expect from conventional client-server based database systems. These indices generally hash a given unique-value/key (e.g., a file name) to a peer address-space (i.e., using IP addresses) and hence cannot perform many queries such as range queries on various data types. For example, a spatial object commonly has an extent and cannot be easily represented by a single point/data value, and popular spatial queries, such as spatial range queries, are not exact match queries. In this paper we focus on the use of P2P networks for applications with spatial data and queries. Such applications are greatly enhanced by the availability of a spatial index which speeds up the retrieval capabilities of the application. In particular, in this paper, we introduce and analyze a distributed spatial index that is based on the quadtree data structure (e.g., [30–32]).

One of the most common spatial queries is the *spatial range query*. In this case, the results of a conventional query are further constrained by being restricted

to a particular spatial region, also known as a *window query* when viewed from a computer graphics perspective. On the other hand, from a database perspective, this query is referred to as a *spatial selection query* [3]. This query can arise in a number of applications ranging from a distributed database of real estate ads (such as those found in many online newspapers) where we want to find all ads involving certain properties in a given region of a city. Clearly, many other variants of this query could be formulated and also in terms of combinations of other attributes of the data. In particular, window queries can also be formulated for nonspatial attributes such as finding all individuals whose height, weight, and age are within a given set of value ranges. In this paper, our focus is on two-dimensional spatial data although the methods that we present are generalizable to higher dimensions. In fact, we have incorporated our methods in a P2P 3D virtual world application where users can update the world without central administration [36]. We have also tested the methods described in this paper using a realistic two-dimensional spatial setting and the results are reported in this paper. These results demonstrate that our distributed spatial index and the associated algorithms work well under a wide range of parameter settings and also exhibit a promising load-balancing behavior.

The rest of this paper is organized as follows. Section 2 gives an overview of our contributions and reviews related work. Section 3 presents our distributed quadtree index in detail. Section 4 reports the results of experiments using our index. Section 5 contains concluding remarks as well as provides directions for future research involving possible extensions of our work.

2 Approach and Related Work

2.1 Approach

We separate the details of the P2P protocol (e.g., [35]) implementation from the spatial index by using a layered approach proposed in [18], called the Open P2P Network (OPeN) architecture. The OPeN architecture consists of three layers; the *Application* layer, *Core Services* layer, and *Connectivity* layer. The Application layer is where all the application logic is confined. The Core Services layer ensures consistency and ease of development for a large range of applications using common services. The Connectivity layer enables P2P protocols to be developed transparently. In particular, the architecture abstracts the method by which the P2P protocol undertakes key-based routing. The performance of our spatial index, named as Spatial Data Service, is dependent on the performance of the protocol, but not on the protocol implementation details. The layered approach allows us to optimize either the spatial index or the protocol independently of one another, if needed. This brings the flexibility to transparently exchange the protocol without changing the spatial index. It also allows the use of such methods as object replication and migration, transparently to the spatial index. Quality of service parameters can be used to influence migration and replication processes. In summary, our index is built on a P2P protocol implementation that can be easily replaced and influenced. The layers of OPeN are shown in Fig. 1.

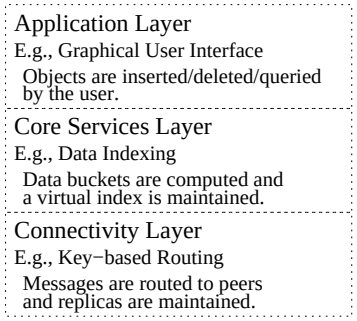

Fig. 1 Our spatial index is implemented in the Core Services Layer, between the Application Layer (top) and the Connectivity Layer (bottom).

Our distributed spatial index assigns responsibilities for regions of space to the peers in the system. Using a quadtree, each sub-region is uniquely identified by its centroid where the recursive space subdivision lines meet. We call this centroid a *control point*. We then pass this information as a key and use a key-based routing protocol (e.g., Chord [35]) at the Connectivity Layer to hash these control points to peers.

2.2 Key-based Routing

In a key-based P2P routing protocol, given a (*key*,*message*) pair and a hash function that maps keys to address locations, the protocol routes the message from any given source peer to the destination peer; the destination is the peer assigned to the key's address location. A hash function is commonly used to randomize the mapping from keys to address locations. Roughly equal, contiguous blocks of address locations are assigned to each peer. Assuming the key represents an object to be accessed according to the message, then the destination peer is responsible for maintaining that object's details in its memory. In some cases the object details may be a reference to some other peer where further details about the object can be found.

We use the Chord protocol [35] as a key-based routing protocol at our base. Other key-based routing protocols exist such as the CAN protocol [27] and can be used with our work.

Fig. 2 shows a simplified example of the Chord routing algorithm. Keys and IP addresses of peers are

mapped to virtual locations in the range from 0 through $2^t - 1$. The shaded area in Fig. 2 shows the range of the

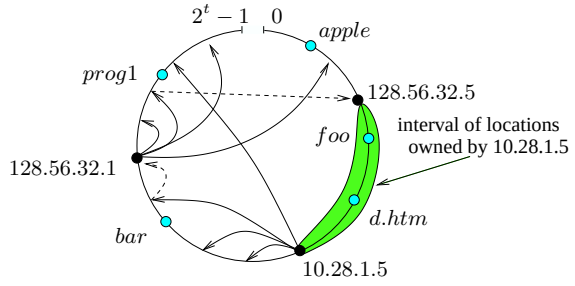

Fig. 2 Details from the Chord method.

address space corresponding to the locations of keys for which peer 10.28.1.5 is responsible. For n peers, each peer maintains $O(\log n)$ routing information, i.e., the routing table size, and it can be shown that a request to locate a file will be forwarded $O(\log n)$ times with high probability. Entries in the routing table define exponentially increasing intervals in the address space and each entry is associated with a peer that occurs next, traversing clockwise, in that interval. Messages continue to be forwarded from peer to peer until the request reaches the destination peer. In Fig. 2, a message with key *prog1* can be routed from 10.28.1.5 to 128.56.32.1 and finally to 128.56.32.5. Only some of the routing table entries are shown in our figure.

2.3 P2P Complex Queries and Complex Data

2.3.1 P2P Range Queries

The implementation of range queries on P2P networks has rapidly emerged as an area of interest. PePer [12] supports range queries in one dimension by subdividing the underlying space into regular intervals. MAAN [9] uses locality preserving hashing to map a range of data space to Chord. This approach uses either a direct mapping of the data domain to the Chord space or assumes that the input data range and distribution are known in advance to create a balanced mapping. Other related work includes [20] which uses directed acyclic graphs to create a range addressable topology and [4,14] which use methods that group data points with skip-graphs [5] or utilize range-partitioning with online balancing algorithms. CLASH [24] has introduced variable-length keys and clustered content-related objects on peers. It can change the length of the keys to adjust for load. Also, [26] uses a trie-based scheme and hashes prefixes of keys to peers. Similar to CLASH, when a certain threshold is exceeded, a split occurs with two new prefixes. P-trees are introduced in [11]. The P-tree concept is very similar to the B+-tree concept but P-trees are decentralized structures. In [17], Gupta et al. use caching and also locality preserving hashing to approximate answers for range queries. Also, in [8], authors introduce a multi-attribute range query system with explicit load-balancing measures. In essence, they use multiple circular overlays and organize the peers of the system into these overlays.

Various approaches (e.g., [2,29]) have been proposed using the CAN [27] key-based routing protocol. A two-dimensional CAN space can be viewed as a grid of cells and the peer addresses and keys for the data can be hashed onto this space. Each peer knows about its four adjacent neighbors and this knowledge is used for routing. Recent approaches for range queries rely on the *direct* mappings of the data space onto the CAN namespace. Such a direct mapping can have load-balancing issues with skewed data distributions. The peers that take ownership of regions of space where there are many data items can suffer from load-balancing problems. Even if these regions are subdivided, the CAN method is based on connecting neighboring cells to facilitate routing of queries in the P2P network. Hence, the neighbors can be congested. More importantly, routing using only neighboring cells takes time for a large network without a method (e.g., a hierarchy) to enable large jumps in the network.

In CAN, virtual peers could be placed in areas of high load and this may be preferred to the case where the load easily spreads to neighboring peers. In fact, if the load is spread to randomly selected peers, then the load balancing is similar to what is achieved using our approach. An interesting observation is: A d-dimensional CAN network has a lookup time of $\frac{d}{2}n^{1/d}$ with $2d$ neighbors at each of n peers. Setting the lookup time to $\log_2 n$ (like Chord) and hence solving $\frac{d}{2}n^{1/d} = \log_2 n$ for d yields $d = \log_2 n$ and this leads to the same topological properties as Chord. This can be used for *indirect* mappings of the data space to the namespace which would be fundamentally equivalent to our work.

In comparison to our index, none of these approaches consider the case where data may have multiple dimensions attached to each other with extents in each of these dimensions, as it is the case with spatial data, and complex queries on this data, such as window queries.

2.3.2 P2P Spatial Data

Recently, Mondal *et al.* in [25] reported on their preliminary work on using a P2P R-tree index. Our approach is different than theirs in that we are using a quadtree spatial index which is based on a regular decomposition of the underlying space. The benefits of our approach are: a quadtree decomposition is implicitly known by all the peers in the system without a need for any communications, and future work can benefit from our work by facilitating operations between separate data sets as the partitions are in registration with each other which

is not the case in R-trees. Also [15] describes two approaches for accommodating window queries. First, they use space-filling curves with range partitioning to reduce multi-dimensional data into one dimension. Next, they utilize skip-graphs for efficient range queries. Ganesan et al. [14] also point out that load-balance can be an issue for such direct mappings and it needs to be fixed with external techniques such as the one described in [14]. In our work, we do not require explicit load-balancing algorithms. Second, they introduce a P2P version of the k-d tree [7] that is similar to the direct CAN approach to address point data. The routing again utilizes the neighboring cells of the data structure. For this second approach, they also argue that load-balance is an issue that needs to be addressed although they leave its resolution for future work. Finally, [6] recently described a method based on using a Voronoi diagram to address multidimensional objects on P2P networks. Their work is similar to the CAN approach. In comparison to a regular, grid like, space division, they subdivide the space using a Voronoi diagram. They use random graphs for routing which can connect remote peers/regions. However, the random graph method does not have the deterministic behavior of a quadtree. Also, Voronoi diagrams are harder to manage with high dimensional data due to, in part, their large space requirements as the dimension of the underlying space increases.

2.3.3 Related Work from Wireless Networks

Recently, [13] introduced an R-tree index in wireless sensor networks. Their work uses explicit cluster heads for maintaining connectivity between peers responsible for parts of the data while in our work we try to decentralize this concept and maintain connectivity more implicitly. They also only focus on performing nearest neighbor queries. The work reported in [21] introduces a location service for ad hoc networks. Although they also use a recursive subdivision of space, their main concern is on locating and routing to point objects in this space rather than facilitating general spatial queries on spatial data. In [22], Li et al. introduce a distributed data structure that depends on locality preserving geographic hashing. They report addressing skewed data as future work and their structure is constructed for addressing concerns related to point data rather than data with extents. Hence, the data structure can be viewed as a distributed k-d tree. In [16], Gao et al. use a quadtree-based scheme to store sensor data. They use this structure to aggregate data over a large area in a fractionally cascaded manner with respect to distance.

In summary, wireless sensor and mobile ad hoc networks are concerned with mostly point data, and algorithms are driven by a desire to utilize the physical network connectivity given certain constraints such as power. Also, mappings between processors and data is commonly bound by the fact that processing units share the same physical space with the data. P2P networks over the Internet are hence fundamentally different than these networks in many aspects, although they have other similarities such as their decentralized nature.

3 Distributed Quadtree-based Hashing

Spatial objects are objects with extents in a multidimensional setting. The generality of the shapes of the objects and the query regions means that the process of intersecting them is more complex than finding exact matches when dealing with file names in the case of documents or music files. Spatial queries are often executed by recursively subdividing the underlying space and then solving possibly simpler intersection problems. This recursive subdivision process lies at the heart of implementations that make use of some variants of the quadtree representation (e.g., [31]). (For our work, we also consider the situation when additional data, such as pictures of a house for a P2P auction network, can be associated with a spatial object. We keep a reference to the original owner peer of the spatial object where the additional data is stored.)

There are many variants of the quadtree data structure, with the region quadtree being the most common. In this case, the underlying two-dimensional square-shaped space is recursively decomposed into four congruent square blocks until each block is contained in one of the objects in its entirety or is not contained in any of the objects. The advantage of the quadtree representation lies in part in reducing the complexity of the intersection process by enabling the pruning of certain objects or portions of objects from the query. In another common subdivision method, for each object o, the decomposition of the underlying space halts upon encountering a block b such that o overlaps at least two of the child blocks of b or upon reaching a maximum level of decomposition of the underlying space. In both cases, the object is associated with b upon halting. This method has been widely used in applications ranging from VLSI design where it is known as an MX-CIF quadtree [19] to spatial databases where it is known as a filter tree [1,33] and in game programming where a variant in 3D is known as a loose octree [37]. We choose MX-CIF quadtrees to exhibit our P2P index and associated algorithms although other quadtree types could have also been utilized.

If we can attach a peer to a region of space, then that peer is responsible for all query computations that intersect that region, and for storing the objects that are associated with that region. To this end, we make the observation that each quadtree block can be uniquely identified by its centroid, named a *control point* and we can use the Chord method to hash these control points so that the responsibility for a quadtree block is associated with a peer. For example, $H(\text{“}(5,2)\text{”})$ is the location of the control point $(5,2)$ on the Chord. The control points

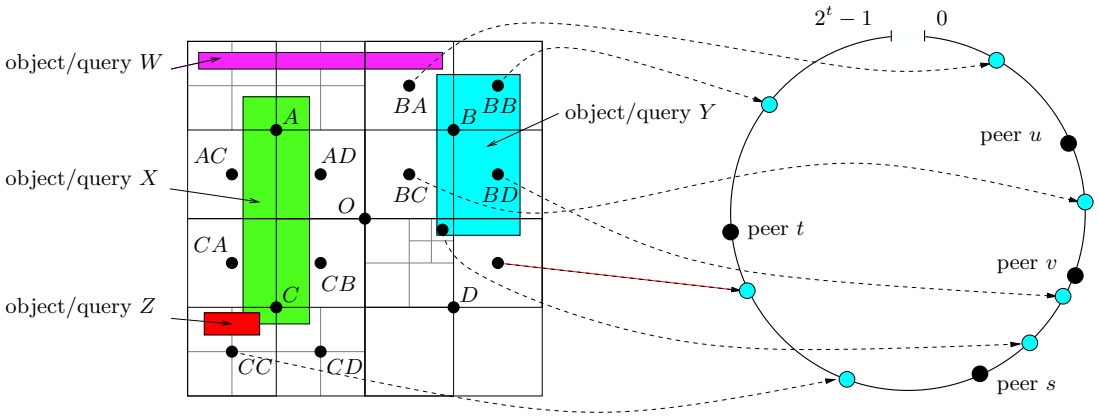

Fig. 3 Spatial objects/queries $\{W, X, Y, Z\}$, control points, and some of the hashings to the Chord, i.e., the coordinate values of a control point are used as the key and hashed onto the Chord. Dark dots are the peers that are currently in the system. Light dots are the control points hashed on to the Chord. For this figure, $f_{min} = 2$.

can be determined using the globally known quadtree subdivision method to recursively subdivide the underlying space. Multiple control points and hence quadtree blocks can be hashed to the same peer and multiple objects can be stored with each control point. The use of a control point in our algorithms is analogous to that of a bucket for storing objects and also for performing intersection calculations associated with that block.

With a good base hash function we can achieve a uniformly random mapping of the quadtree blocks to the peers of the network. For example, SHA-1 is a good candidate hash function that is used with Chord. It will map two quadtree blocks that are close to each other, as well as two peers with similar IP addresses, to totally different locations on the Chord space. Hence, as quadtree subdivisions create new blocks for crowded regions of the space, we will be assigning the responsibilities of these blocks to different peers, creating an implicitly load-balanced method that can handle skewed data distributions. A perfect load-balance for our purposes is defined as having all of the peers in the network sustain an average load throughout their existence in the network. The average load can be defined as processing an average number of messages. Thus a good load-balance is when very few peers process many more messages than the average number.

Fig. 3 depicts some control points and example hashings using the Chord method. Objects are inserted into the distributed structure by mapping them into quadtree blocks and hashing the control points of those blocks on to the Chord. In the example, control point CC is mapped to peer t and the object Z is stored with that control point.

At first glance, query execution starts at the root of the quadtree and propagates down through some branches of the tree, testing for the intersection of data objects with the query object as it proceeds. For a P2P system, the tree traversal becomes a sequence of peer visits. The query is transmitted from a parent block b in the quadtree (i.e., from the peer to which the control point maps) to the child blocks (i.e., to the peers that store the child blocks) by utilizing the Chord P2P lookup method.

Unfortunately, there is a single point of failure that occurs when all tree operations begin at the peer that stores the control point associated with the root. If that peer is very busy or the ownership of that control point has to change hands with peer departures, then the whole tree will become unavailable for some time. On the other hand, as we have a distributed structure, there is no reason for us to start the operations at the root of the tree. Therefore, we introduce the concept of the *fundamental minimum* level, f_{min}. This modification means that objects can only be stored at levels $l \geq f_{min}$, and likewise all query processing occurs at levels $l \geq f_{min}$. Hence, for our quadtrees, no objects can be stored at levels $0 \leq l < f_{min}$. For example, in Fig. 3, f_{min} is set to 2. Hence, the object X is subdivided at level 0 into an upper and a lower part, and then further subdivided at level 1 into 8 different parts. 4 out of the 8 parts of X are stored at level 2 (i.e., associated with control points AC, AD, CA, and CB, respectively). Note that these 4 parts cannot be stored at a deeper level of the tree without having to be subdivided again, which would contradict the original definition of an MX-CIF quadtree. The remaining 4 parts are inserted at the next (deeper) level as they are smaller. In particular, they are stored at control points AAD, ABC, CCB, and CDA, respectively, although, in the interest of keeping the figure simple, they are not shown in detail. Again, these parts cannot be inserted at a deeper level of the tree as this would require that they be subdivided. As another example, the smallest part of object Y is inserted at level 4.

Note that the concept of f_{min} also helps distribute the initial processing of a query to lower levels and hence implicitly further balance the load. In addition, we continue to use the concept of a maximum depth for a MX-CIF quadtree and name it as the *fundamental maximum* level, f_{max}. Values for f_{max} and f_{min} are constant and globally known. When $f_{min} = 0$, our structure reverts to the standard MX-CIF quadtree, but a distributed version of it. When $f_{min} = f_{max}$, our structure degenerates to a distributed regular grid structure thereby completely collapsing the tree structure into a single level.

3.1 Distributed Spatial Algorithms

Our basic operations include:

1. When an insertion operation or query is initiated at a peer, the peer calls the InsertObject() or ReceiveClientsQuery() which in turn calls Subdivide() to compute the intersecting control point(s) at the f_{min} level. It then broadcasts the insertion operation or the query to the peer(s) that owns the control point(s).
2. When a peer receives an insertion operation or query that was initiated on another peer, it calls either DoInsert() or DoQuery() which determines the placement of the spatial object or finds any relevant results, respectively, and finally decides whether the operation needs to continue by descending through the distributed tree.

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 objects that exist at or below child i. $list$ is a list of objects that intersect the region $R(u)$ and that could not be inserted at a deeper level in the tree. The default values for $D(u)$ are $(\{0,0,0,0\}, empty)$ which means that there are no objects below u and no objects stored at u. We do not create a control point until an insertion operation requires it. Throughout the paper, $R(u) = (x_1, y_1, x_2, y_2)$ denotes the region defined by a quadtree block centered at control point $u = \big(((x_2 + x_1)/2), ((y_2 + y_1)/2)\big)$. Also, we let $L(u)$ denote the level or depth of the quadtree at which control point u is present, and $C(u, i)$ to represent the i^{th} child of control point u, where $i = 1, 2, 3, 4$. Finally, $D(u).field$ is used to access $field$ of $D(u)$.

Operations on our index are decentralized. We use Delegate(u)→Func() to mean that a peer sends a message that invokes Func() on another peer that stores control point u. With delegation, operations can propagate through the tree. They work in parallel on multiple branches on different peers.

A peer calls InsertObject(object X) to insert an object into the tree. Procedure InsertObject(), given below, concurrently delegates procedure DoInsert() over all control points at level f_{min} that intersect with the object. Procedure DoInsert() delegates recursively and also concurrently through the distributed tree until the object is inserted. If DoInsert() is invoked on a control point that does not exist, then the control point is allocated with default parameters.

Our algorithms make use of auxiliary procedures $Ints(X, Y)$ and Subdivide($X, root, G$). $Ints(X, Y)$ computes the intersection of X with Y. Subdivide($X, root, G$) is initially called with $G = \{\}$ when inserting X (also used when deleting or starting a query). This initial subdivision of an object (or query) down to level f_{min} is performed by the recursive invocation of Subdivide(): The procedure modifies G by adding control points to it. First, $root$ is used to call this procedure where $L(root) = 0$ and $R(root)$ is a bounding box that bounds all data and query objects. For this algorithm, it is assumed that all X's are contained within $R(root)$. The list G of control points at level f_{min} is then computed locally on the peer that will start the insertion operation, and processing is then delegated to the peers that store these control points.

```
InsertObject(object X) {

  control point list G:={}
  Subdivide(X, root, G)
  for each u in G do in parallel
    Delegate(u)→DoInsert(X, u)
}

DoInsert(object X,
         control point u) {

  if (X is not within exactly one R(C(u,i)))
   or (L(u) = f_max) then
   set D(u).list to include X
  else
    for i:=1 to 4 do in parallel
      if (Ints(X, R(C(u,i))) is not empty) then
       increment D(u).d_i by 1
       Delegate(C(u,i))→DoInsert(X, C(u,i))
}

Subdivide(object X,
          control point u,
          control point list G) {

  if (L(u) = f_min) then
   add u to G
   return
  for i:=1 to 4 do sequentially
    if (Ints(X, R(C(u,i))) is not empty) then
     Subdivide(X, C(u,i), G)
}
```

Procedures for DeleteObject() and DoDelete() are almost identical to InsertObject() and DoInsert() and thus are not given here. The main difference is that objects are removed from $D(u).list$ and that $D(u).d_i$ is decremented by 1.

We allow peers to receive a query from any node on the Internet (i.e., a client may or may not be a peer in the system) via the ReceiveClientsQuery() procedure. Similar

to insertion (deletion), this procedure takes in an object (named Q for query in this case) and then finds the control points at level f_{min} and delegates the query to the relevant peers. The results of a query can then be sent back to the client. This requires that a message from any peer that computed a piece of the query intersection be sent to the peer that stores/owns the extra data related to the hit which in turn sends this data to the client.

```
ReceiveClientsQuery(query Q) {

  control point list G:={}
  Subdivide(Q, root, G)
  for each u in G do in parallel
    Delegate(u)→DoQuery(Q,u)
}

DoQuery(query Q,
        control point u) {

  intersect objects in D(u).list with Q
  send results to client
  for i:=1 to 4 do in parallel
    if (Ints(Q,R(C(u,i))) is not empty)
       and (D(u).d_i > 0) then
      Delegate(C(u,i))→DoQuery(Q,C(u,i))
}
```

Note that in Fig. 3, Y is bucketed in 6 different control points that span 4 different peers. This means that a query which covers these multiple control points may return the same object a multiple number of times, and thus we have to eliminate such superfluous hits. In fact, it is more important to prevent repetitive large downloads to a client from the owner of the object. The peer who is the original owner of the object and the extra data associated with it is responsible for this elimination task. Also, for concurrency control and tracking the progress of the tree operations, this owner peer can again be used.

Delegation is achieved using the Chord method, and thus it ordinarily takes $O(\log n)$ messages for the delegation to reach its destination where n is the number of peers in the system. Since each node of our distributed MX-CIF quadtree has a fixed number of children, we can allow each node to maintain a cache of addresses for its children and thereby reduce the delegation message complexity to $O(1)$. This is only true when the operation is a regular tree traversal. Note that the number of peers that are initially contacted is a function of f_{min} and can consist of a large number of addresses. Hence, we do not cache the addresses of the peers of level f_{min}. When the set of peers in the P2P network changes, the server tables of the peers and the ownership of control points are updated by the Chord method. However, with this change, our caches may also need to be updated. We delay such updates until there is a cache miss. An addition to our caching strategy could be to propagate the positive results to the f_{min} level although this is not investigated with our current implementations.

3.2 Index Latency and Load Balancing

We assume that the latency of sending a delegation message from one peer to another peer is the dominant factor in our index, e.g., in comparison to internal processing costs of peers. We also assume that the different branches of a single query can proceed in parallel through the system. Therefore, under light load the maximum index latency is the time for a longest branch of the distributed quadtree to be traversed. By light load we mean that the time taken for a message to be processed and forwarded by a peer is independent of the number of messages received and sent by the peer. Under a heavy load this is not true because each peer has limited bandwidth and messages will suffer delays due to queuing. For n peers, under light load, index latency is proportional to the number of message hops on the longest path:

$$O(\log n + f_{max} - f_{min}).$$

Note that the $O(\log n)$ component cannot be avoided because the root(s) of the tree(s) must be located regardless of the value of f_{min} using the underlying Chord method. For $f_{min} = 0$, we have to locate the root of the distributed quadtree and traverse the tree until the maximum allowable depth of the tree, f_{max}, is reached. The traversal of the tree itself does not need to use the Chord method due to caching. This traversal would then be analogous to marching along the longest path of a classical quadtree on a centralized system where memory accesses rather than messages are used. A quick analysis suggests that $f_{min} = f_{max}$ should be the most optimal value for a minimum index latency. This would also mean that all the queries will not go through a single root and hence we would increase the load balance in the system. Unfortunately, increasing f_{min} to be equal to f_{max} has its side effects. The pruning capability of the tree would diminish and we would have a regular grid with increased load. Hence, given a quadtree, with an f_{max}, finding the right f_{min} value is our goal.

From a load balancing point of view, to find a value for f_{min}, we first ask what value of f_{min} ensures that all peers in the system contribute to the query processing at level f_{min}. If only a fraction of the peers at level f_{min} participate, then some peers will potentially have larger message loads than others. The number of control points at level f_{min} is $4^{f_{min}}$ and these control points are distributed uniformly at random over the peers using a hash function. From basic probability theory, it is known that if we consider m balls that are to be distributed uniformly at random over n bins, then it can be shown that the average number of bins that will receive at least one ball is:

$$\phi(n, m) = n - ((n-1)^m \cdot n^{1-m}).$$

The intuition behind this formula is that the probability that a bin receives none of the m balls is $((n-1)/n)^m$. Therefore, the probability that a bin receives at least one

of the m balls is $1 - ((n-1)^m/n^m)$, which is multiplied by n to obtain the average number of the n bins that will receive any of the balls. When throwing only $m = n$ balls,

$$\lim_{n\to\infty} \frac{\phi(n,n)}{n} = \frac{e-1}{e} \approx 0.63$$

while for $m = n \log_2 n$ balls,

$$\lim_{n\to\infty} \frac{\phi(n, n \log_2 n)}{n} = 1.$$

Thus we can write, with constant c_0:

$$\begin{aligned} 4^{f_{min}} &\geq c_0\, n \log_2 n \\ f_{min} &\geq \log_4(c_0\, n\, log_2 n) \\ f_{min} &= \Omega(\log n + \log\log n). \end{aligned} \tag{1}$$

So long as f_{min} meets the requirement in Eq. (1) then each of the peers in the system has an equal opportunity to handle the message load at level f_{min}. For a uniform distribution of queries over the data space, this is the minimum value of f_{min} required to make sure that load is distributed in an unbiased manner over the peers. For skewed data distributions, higher values of f_{min} may be necessary to achieve a perfect load distribution.

From an overall load point of view, the magnitude of f_{min} is limited from above. For higher values of f_{min}, queries will be unnecessarily subdivided and sent to multiple peers. Assuming that the average query rectangle size is a constant fraction ξ of the space in each dimension, then ξ^2 is the fraction of the total space that a query covers. In this case, the number of control points at level f_{min} that are covered by a query is no more than

$$\left(\lceil \xi\, 2^{f_{min}} \rceil + 1\right)^2 < \left(\xi\, 2^{f_{min}} + 2\right)^2.$$

Therefore, we can maintain a $O(\log n)$ messages per query at the initial lookup phase for f_{min} level peers so long as

$$f_{min} = O(\log 1/\xi),$$

which gives an upper bound on f_{min} if we do not want to increase the message load in the system. Higher values of f_{min} will start dividing the queries into smaller subqueries.

For example, for $n = 1000$ peers (and setting all constants to 1), we need f_{min} to be at least 7 for a perfect load balance. However, for $\xi = 0.04$, f_{min} should not be larger than 4 in order to ensure that there is no increase in message load due to the value of f_{min}.

4 Experiments

4.1 Experimental Environment

We used the Network Simulator, ns-2 (www.isi.edu/nsnam/ns), in tandem with the Georgia-Tech Internetwork Topology Generator, GT-ITM (www.cc.gatech.edu/projects/gtitm) in our experiments. Each experiment examines a different aspect of our index and consists of multiple observation points. Each observation point is obtained after 5 consecutive runs with the same input parameters given to the simulation environment. We averaged these 5 runs to obtain an observation point value. Before each run, we created a transit-stub type network at random using GT-ITM (Fig. 4). Each transit domain can be considered as representing a different metropolitan area network. Transit nodes can be considered as the main Internet service provider nodes in the metropolitan areas. Stub domains can be considered as representing different campus or company or such similar entity networks. A stub node is used to represent a small local area network. We did not allow for extra transit to stub edges or stub to stub edges across domains. The links between transit domains had a lower capacity than regular transit nodes in order to properly represent the most congested links in such networks. A summary of values used with ns-2 and GT-ITM is available from Table (1) All of these parameter values were chosen to be comparable to the ones used in other similar studies that deployed ns-2 with GT-ITM [10].

Parameter	Value
No of transit domains	2
Transit nodes per domain	8
Stub domains per transit node	6
Stub nodes per stub domain	12
Probability of connecting transit nodes	0.6
Probability of connecting stub nodes	0.2
Bandwidth between stub nodes	2.5Mbps
Stub to transit node bandwidth	2.5Mbps
Bandwidth between transit nodes	80Mbps
Bandwidth between transit domains	40Mbps

Table 1 Summary of parameters used to generate our simulated network topology.

We have placed our peers randomly on the stub nodes. There is at least one spatial object associated with each peer. For our experiments, we considered clients to be separate from the P2P network, and they were also placed randomly on the stub nodes. For a given scenario, a number of clients arrive, with a single query each, as a flash crowd to the P2P network.

Our simulated networks were generated in the context of a P2P real estate application. The data was generated using US Census Bureau data on postal codes over the Washington, D.C. and Baltimore metropolitan

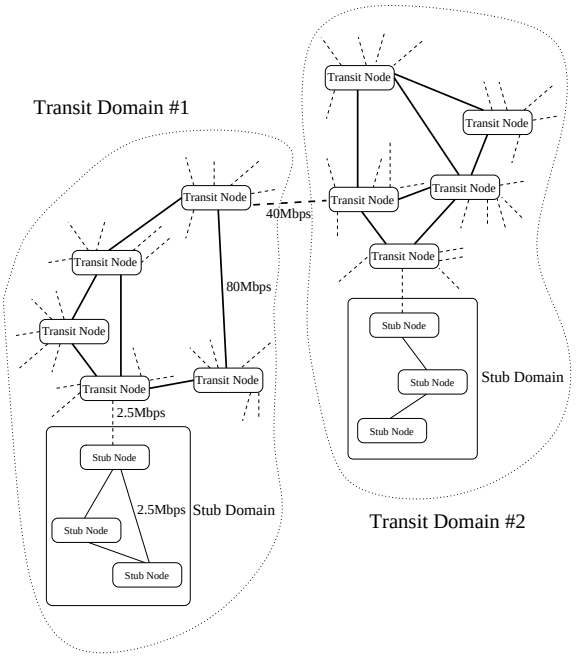

Fig. 4 An example transit-stub type network.

Fig. 5 A set of queries submitted to a P2P network. The data is for the Washington, D.C.– Baltimore corridor.

corridor. We have generated small rectangles representing houses and land using the given population distribution over postal codes. Queries were generated using the centroids of the regions corresponding to the postal codes, the population distribution information, and the area size information on the area of the two metropolitan cities and their surrounding suburbs. A sample set of queries is shown in Fig. 5. The clients were assumed to be making area selections from a map for searching houses or land for sale or rent. Similar applications are commonly available from online newspapers where users can enter the postal codes in text form for their queries. Each object also had some extra data, e.g., a picture of the house/land for sale, attached to it.

The simulation environment is controlled by using the following parameters: The number of queries that formed the flash client crowd, the number of data objects that are in the P2P network, the number of peers in this network, the query rectangle size (as a parameter that varies the original postal-code-based query rectangle size), the data size attached to an object, f_{min}, f_{max}, and the percentage value of misses in the system for our caching scheme.

4.2 Results

4.2.1 Changing f_{min}

We first varied the f_{min} parameter value to see how our index behaves with increasing f_{min} values. The hypothesis is that with increasing f_{min} values we will first see a decrease in the index latency and also a better load balance among peers. However, eventually we should see an increase in the overall load of the system.

For this experiment, there are 1000 peers in the network hosting 1000 spatial objects, and the amount of data attached to these objects is assumed to be 100 KBytes. The f_{max} value is 10 and our cache is assumed to have no misses. The query rectangle sizes are kept equal to the postal code area information without scaling. The number of queries that come as a flash crowd is 20.

As f_{min} increases (Fig. 6), we do observe a gradual decrease in the average query processing time for a query. The number of levels of the distributed MX-CIF quadtree that need to be traversed decreases with each increase in the value of f_{min}. Finally, the last few observation points for the f_{min} experiment shows an increase. This is due to increased load. The message counts for this experiment is given in Fig. 7. As expected, this graph has a very steep increase in the number of messages per query as f_{min} reaches its peak. The increase is due to the increase in the number of initial f_{min} level messages per query and hence due to the loss of the pruning capability of the distributed MX-CIF quadtree when it becomes more and more like a regular grid.

When we analyzed the trace data for this experiment, we saw that there were actually only a few queries that create the drastic increase in query processing time.

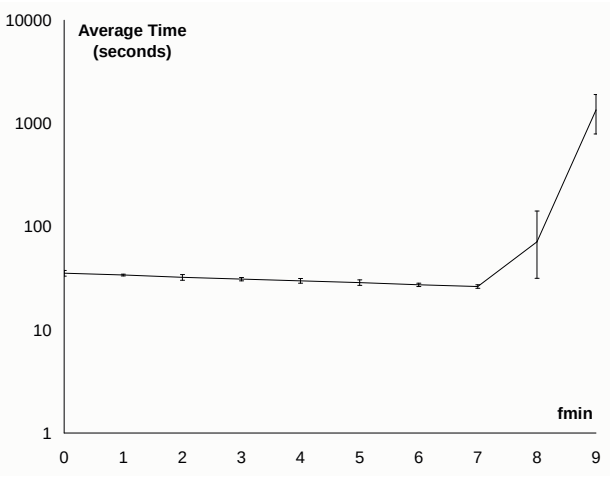

Fig. 6 Average query processing time as f_{min} increases (standard deviations are also shown).

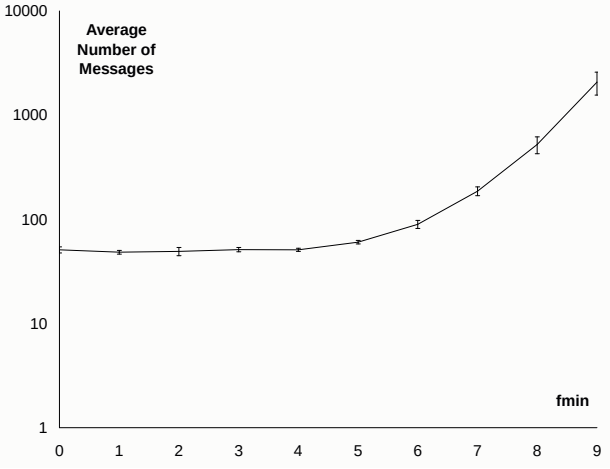

Fig. 7 Average number of messages per query as f_{min} increases.

These queries are from the regions of space where almost no data objects exist but we saw that the query rectangle size in these regions is much larger than the average query rectangle size. These are the areas of the postal codes corresponding to the outer suburbs which were found to contain very few houses or land for sale. The pruning capability of the distributed MX-CIF quadtree in these cases is very important. Otherwise, i.e., with a large f_{min} value, a large number of messages for f_{min} level nodes can be generated for these queries. Hence, the peers that initiate these queries started to fail first under the load.

We can also see the change in the load distribution for this experiment by looking at Fig. 8. This graph looks at the standard deviation in normalized load for different f_{min} values. As expected, we see a decrease first, as the number of peers contributing at level f_{min} increases, with increasing values of f_{min}. Later, we see an increase due to the fact that the query initiating peers started to generate many redundant messages that overload these peers.

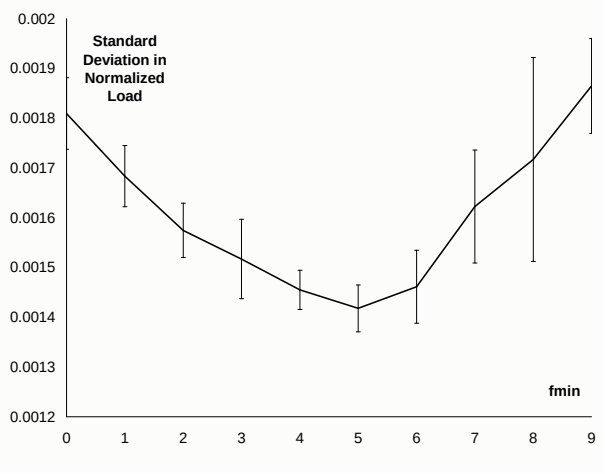

Fig. 8 Standard deviation in normalized load for different f_{min} values.

Fig. 9 provides the actual load per peer. In this figure, bars indicate the number of peers that observe a certain load indicated by the buckets on the x-axis. Each bar for a given bucket maps to an f_{min} value from the range 0 to 9 (left most bar maps to $f_{min} = 0$). The first bucket indicates how many peers with up to (and including) 5 messages existed in the system. The second bucket indicates the number of peers with 6 to 10 messages. The third corresponds to peers with 11 to 20 messages. The rest of the buckets are organized in a logarithmic manner. As the value of f_{min} increases, we see a gradual improvement in participation at f_{min} level. For example, for small values of f_{min} there are a few peers with more than 20 messages. Then, for larger values of f_{min}, we see that no peers do actually process more than 20 messages. Later, with a further increase in f_{min}, we see an increase in load and many peers start serving more than 40 messages, many of which are redundant messages as the tree started to lose its pruning capability.

4.2.2 Elasticity

We also wanted to run experiments to see how our index behaves with changing values of the parameters such as the number of peers in the system. The following gives the results of these experiments.

The first elasticity experiment gives a base in scale and runtime behavior for the distributed MX-CIF

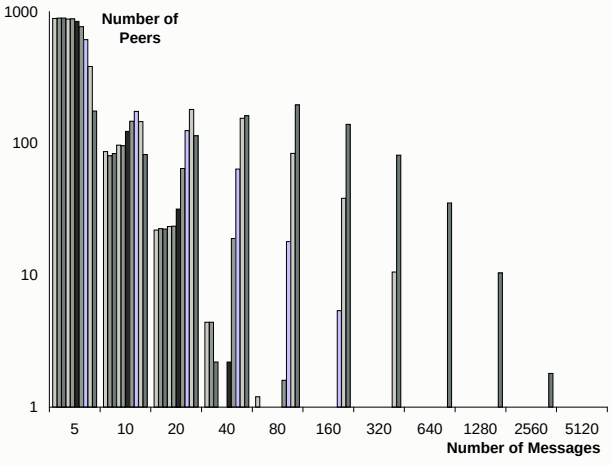

Fig. 9 Load for different f_{min} values.

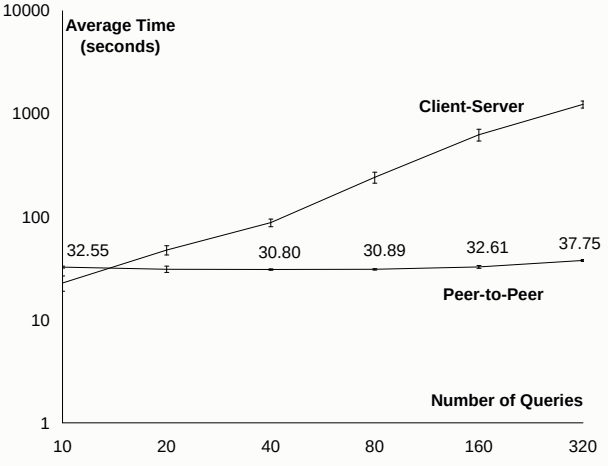

Fig. 10 The average time per query for the distributed MX-CIF quadtree index as the number of queries increases, in comparison to a central index. The numbers on the line belong to the distributed MX-CIF quadtree index and they are the corresponding y-axis values.

quadtree index in comparison to a client-server system (Fig. 10) by increasing the number of queries given to the system. There are 1000 peers in the network hosting 1000 spatial objects, and the amount of data attached to these objects is assumed to be 100 KBytes. The f_{min} value is 3 and the f_{max} value is 10. For this experiment, our cache is assumed to have no misses. The query rectangle sizes are kept equal to the postal code area information without scaling. As expected, the distributed MX-CIF quadtree index scales well in comparison to a client-server system. The time needed to download a large number of hits determines the performance when we have a large number of queries. The P2P system basically has a vast amount of unused bandwidth that easily dominates a server bottleneck.

Second, we see how the distributed index scales with the increasing number of peers in the system (Fig. 11). The number of queries is fixed at 100 for this experiment. The number of objects is 3000. The remaining experiment parameters have the same values as in the first elasticity experiment. We also altered the cache miss ratio to 5, 10, and 15 percent for our caching scheme and repeated the experiment. This was done in order to simulate the fact that some peers in the P2P system may have disappeared (or just appeared) and hence some of the cached information became outdated. The part of our index that is to be most affected by increasing the number of peers is the part where we first connect to the f_{min} level peers. We use $O(\log n)$ steps for this, for n peers, and hence this should have a negligible effect on the performance. From Fig. 11, we see that the distributed MX-CIF quadtree index scales well with the increasing number of peers. Also, for the cases where the cache miss ratio is 5, 10, and 15 percent, there is only a slight change in the behavior of the index.

Third, we see how the distributed index scales as the size of the window query increases (Fig. 12). The number of queries is fixed at 100 for this experiment. The rectangular query objects are originally determined by using the postal code data. We later scale these values to increase the rectangle size by a given factor in each dimension for the different observation points of the experiment. The remaining experiment parameters have the same values as in the first elasticity experiment. Fig. 12 shows that the average query processing time increases linearly as the size of the rectangle query object. This is due to the increase in the number of hits found on the average for a query, and the fact that there is more data

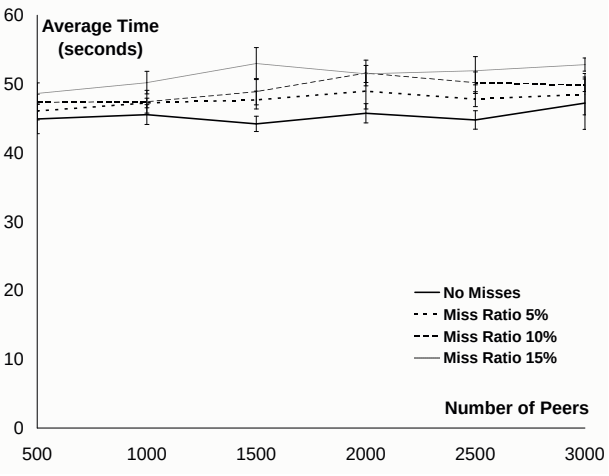

Fig. 11 The average time per query for the distributed MX-CIF quadtree index as the number of peers in the system increases. There are multiple lines in this graph, for experiments with 0, 5, 10, and 15 percent cache miss ratios.

to be downloaded by the clients from the P2P network (Fig. 13). Note that the distributed MX-CIF quadtree index works in parallel. Hence, although there is also a sharp increase in the number of messages in the system (Fig. 14) as the size of the rectangle query object increases, these messages are processed in parallel. The depth of the tree searches does not change. The quadtree is totally independent of the query objects. Hence, as we cover more space in parallel, we can find many hits without increasing the query time. The increased load on the P2P system, due to the increase in the number of messages for this experiment is negligible.

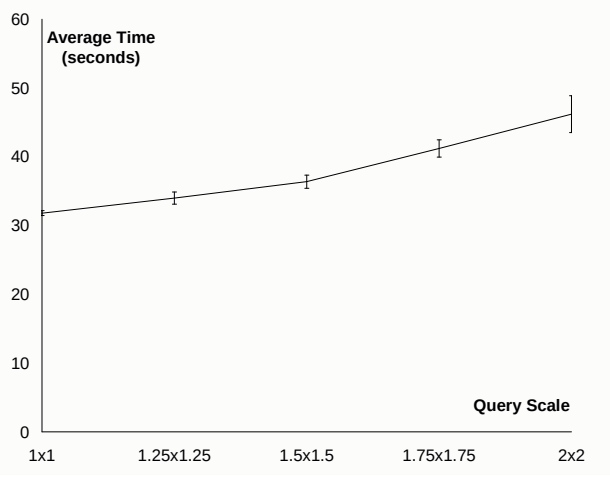

Fig. 12 Average query processing time as the size of the rectangle query object increases.

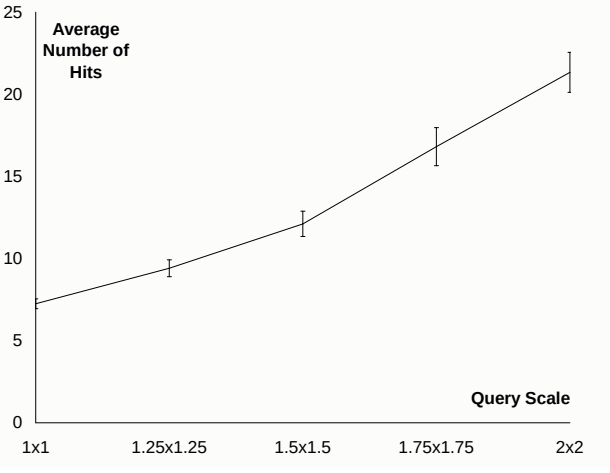

Fig. 13 Average number of hits per query as the size of the rectangle query object increases.

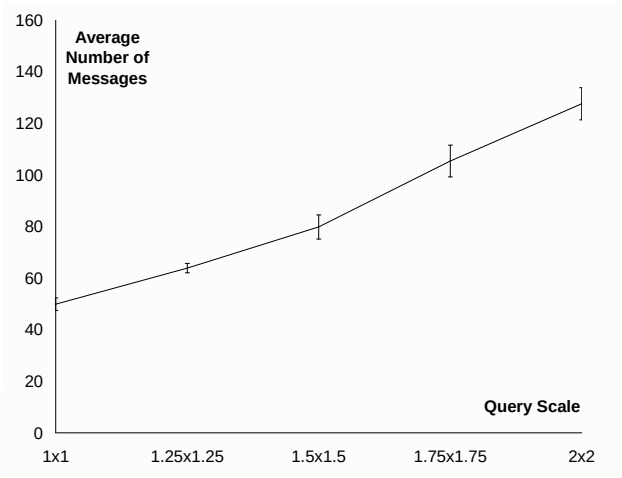

Fig. 14 Average number of messages as the size of the rectangle query object increases.

Given our assumption on the size of each hit (i.e., 100KBytes), for 7 hits (from Fig. 13), we have 700 KBytes of data to download. For 21 hits this increases to 2100 KBytes. Although the downloads also occur in parallel, the client bandwidth cannot be more than 2.5 Mbps on average. The degradation on the client's connection queue dictates the slope of Fig. 12. Confirming this result, we also observed the same result with additional experiments (not reported here) where we varied the sizes of the data attached to the spatial objects which, of course, had the same effect of increasing the size of the downloads. The increase in the data size per object cannot have an effect on the way that the object is searched and located by a query but the experiment shows the same level of degradation in time due to downloads. The bottleneck on the client's connection has no immediate solution.

Fourth, we ran an experiment where the parameter that was varied, was the number of spatial objects in the P2P system and obtained similar results to those obtained when varying the query rectangle size parameter. Fig. 15 shows a linear increase in time as the number of objects per peer increases. The number of peers and queries are fixed at 100 while the rest of the parameters are the same with the first elasticity experiment. For this experiment, we chose a setting where members of the network maintain an order of magnitude more number of objects than the other experiments. This setting is comparable to a setting where a network of real-estate agents, rather than the sellers themselves, form a P2P system.

Our final experiment considered the effects of f_{max} (Fig. 16 and Fig. 17) which is commonly chosen in classical MX-CIF quadtrees to limit the depth of the tree. This experiment also used the same parameter values as the first elasticity experiment but the number of queries

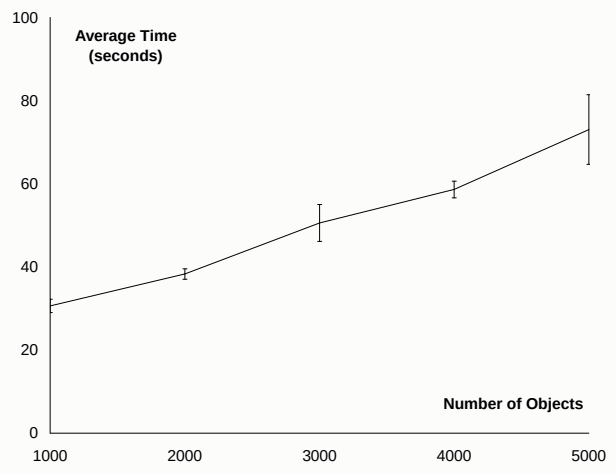

Fig. 15 Average query processing time as the number of objects increases.

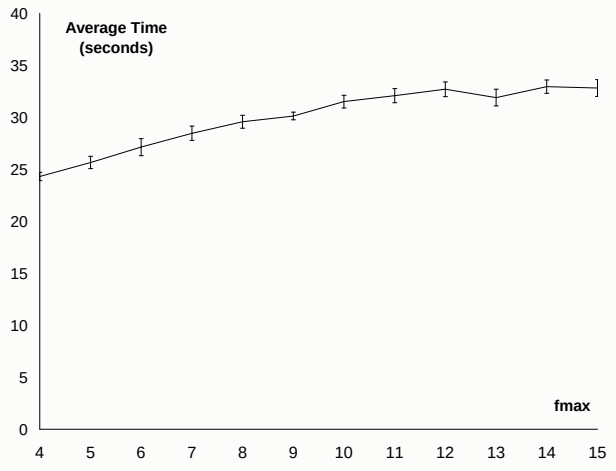

Fig. 16 Average query processing time as f_{max} increases.

was fixed at 100. As expected, as we increased the value of f_{max}, we first observed a gradual increase in the query processing time as there are more levels in the distributed MX-CIF quadtree to be traversed. This was followed by a saturation as the objects (using a distributed MX-CIF quadtree) cannot be inserted at any greater depth in the tree. Therefore, having a very large f_{max} value is unnecessary. In comparison to the rate of increase in the number of messages as f_{max} increases, the rate of increase in the query processing time is much lower as most of the new messages are processed in parallel and only an increase in the height of the distributed MX-CIF quadtree can have a dramatic effect on the execution time. In fact, having a shallow tree may be beneficial in some applications. Nevertheless, a very shallow tree where only a few peers are used to store the entire distributed MX-CIF quadtree will again result in the undesirable presence of a single point of failure and load-imbalance as it is the case with f_{min}.

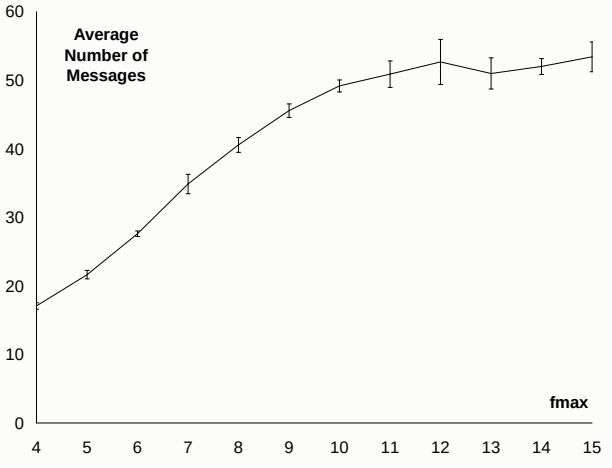

Fig. 17 Average number of messages per query as f_{max} increases.

5 Conclusions and Future Work

P2P networks have become a common form of online data exchange. However, users cannot perform many types of queries on many attributes of the data on such networks other than mostly exact-match queries on complete files. In this paper, we introduced and analyzed a distributed MX-CIF quadtree index to address this problem for spatial data and queries. Our work can be applied to higher dimensions by using high dimensional control points. It can be applied to various data types, i.e., other than spatial data. We can also use different types of quadtrees, i.e., other than MX-CIF quadtrees. Finally, we can use other key-based lookup methods than the Chord method as our base P2P routing protocol. Our experiments show that our algorithms and index work well under many circumstances. The index benefits from the underlying hashing-based methods and can achieve a nice load distribution among peers.

One of the applications for our index is a P2P 3D virtual world application [36]. A virtual world is a representation of objects, along with their relationships in a 3D setting. Users can participate in the virtual world by locating themselves within it, rendering a view of the world, and manipulating the objects in this world. Users may themselves be represented in the virtual world as objects. We are currently using our distributed index with our P2P 3D virtual world application towards implementing a P2P multi-player online game.

For future work, we plan to experiment with a number of extensions to our work. Currently, each of the tree operations we presented starts at a minimum level of the distributed index and proceeds downwards. An alternative approach is to allow tree operations to begin at any level of the tree, i.e., starting operations at random nodes of the tree and allow for bidirectional propagation of operations. Also, we do not need to restrict the size of the underlying space in which the distributed MX-CIF quadtree is built. We can permit object insertions to occur out of the initial limits defined by the starting point of the application. In this case, we must modify our original algorithms so that if an object that is not encapsulated within the initial space arrives, then the peer that handles this request has to spiral out to encapsulate the new object. Another direction for future research is dynamically adjusting the f_{min} parameter. Depending on the workload, data can be pushed downwards or upwards in the tree. But without global communication, access to this part of the space may remain through f_{min} level control points and their dedicated peers. To avoid this, again, starting operations at a random level can be used. Also, a simple binary search over 0 to f_{max} can help find the new f_{min} value. Finally, peers can enter and leave the system at any time and the underlying P2P routing protocol, Chord in this case, is responsible for the transparent handling of these events, but an issue of concern is the lack of application level semantics represented at the protocol level. For example, f_{min} level control points are important and need special attention for replication. One strategy we plan to utilize to safeguard these control points is to pass a quality of service parameter to the protocol layer and increase the level of replication for the entry level of the quadtree. Recently, a similar approach was taken for the TerraDir distributed directory service [34]. Again, starting operations at a random level may reduce or obviate the need for such an extension.

References

1. Aboulnaga, A., Naughton, J.F.: Accurate estimation of the cost of spatial selections. In: Proceedings of the 16th IEEE International Conference on Data Engineering, pp. 123–134. San Diego, CA (2000)
2. Andrzejak, A., Xu, Z.: Scalable, efficient range queries for Grid information services. In: Proceedings of the IEEE International Conference on Peer-to-Peer Computing, pp. 33–40. Linkoping, Sweden (2002)
3. Aref, W.G., Samet, H.: Extending a DBMS with spatial operations. In: Proceedings of Advances in Spatial Databases, SSD'91, pp. 299–318. Zurich, Switzerland (1991)
4. Aspnes, J., Kirsch, J., Krishnamurthy, A.: Load balancing and locality in range-queriable data structures. In: Proceedings of the Symposium on Principles of Distributed Computing, pp. 115–124. St. Johns, Canada (2004)
5. Aspnes, J., Shah, G.: Skip graphs. In: Proceedings of SODA, pp. 384–293. Baltimore, MD (2003)
6. Banaei-Kashani, F., Shahabi, C.: SWAM: A family of access methods for similarity-search in peer-to-peer data networks. In: Proceedings of the Conference on Information and Knowledge Management-CIKM, pp. 304–313. Washington, DC (2004)
7. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM **18**(9), 509–517 (1975)
8. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: Supporting scalable multi-attribute range queries. In: Proceedings of the ACM SIGCOMM'04, pp. 353–366. Portland, OR (2004)
9. Cai, M., Frank, M., Chen, J., Szekely, P.: MAAN: A multi-attribute addressable network for Grid information services. In: Proceedings of the International Workshop on Grid Computing, pp. 184–191. Phoenix, AZ (2003)
10. Cheng, W.C., Chou, C.F., Golubchik, L., Khuller, S., Wan, Y.C.: Large-scale data collection: a coordinated approach. In: Proceedings of the IEEE InfoCom'03, pp. 218–228. San Francisco, CA (2003)
11. Crainiceanu, A., Linga, P., Gehrke, J., Shanmugasundaram, J.: Querying peer-to-peer networks using P-Trees. In: Proceedings of the ACM SIGMOD'04, WebDB Workshop, pp. 25–30. Paris, France (2004)
12. Daskos, A., Ghandeharizadeh, S., An, X.: PePeR: A distributed range addressing space for P2P systems. In: Proceedings of the International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (held in conjunction with VLDB), pp. 200–218. Berlin, Germany (2003)
13. Demirbas, M., Ferhatosmanoglu, H.: Peer-to-peer spatial queries in sensor networks. In: Proceedings of the IEEE International Conference on Peer-to-Peer Computing, pp. 32–39. Linkoping, Sweden (2003)
14. Ganesan, P., Bawa, M., Garcia-Molina, H.: Online balancing of range-partitioned data with applications to peer-to-peer systems. In: Proceedings of the International Conference on Very Large Databases-VLDB, pp. 444–455. Toronto, Canada (2004)
15. Ganesan, P., Yang, B., Garcia-Molina, H.: One torus to rule them all: Multidimensional queries in P2P systems. In: Proceedings of the ACM SIGMOD'04, WebDB Workshop, pp. 19–24. Paris, France (2004)
16. Gao, J., Guibas, L.J., Hershberger, J., Zhang, L.: Fractionally cascaded information in a sensor network. In: Proceedings of the IPSN'04, pp. 311–319. Berkeley, CA (2004)
17. Gupta, A., Agrawal, D., El Abbadi, A.: Approximate range selection queries in peer-to-peer systems. In: Proceedings of the First Biennial Conference on Innovative Data Systems Research. Asilomar, CA (2003)
18. Harwood, A., Karunasekera, S., Nutanong, S., Tanin, E., Truong, M.: Complex applications over peer-to-peer networks. In: Poster Proceedings of the ACM Middleware'04, p. 327. Toronto, Canada (2004)
19. Kedem, G.: The quad-CIF tree: a data structure for hierarchical on-line algorithms. In: Proceedings of the 19th Design Automation Conference, pp. 352–357. Las Vegas, NV (1982)
20. Kothari, A., Agrawal, D., Gupta, A., Suri, S.: Range addressable network: A P2P cache architecture for data ranges. In: Proceedings of the IEEE International Conference on Peer-to-Peer Computing, pp. 14–22. Linkoping, Sweden (2003)
21. Li, J., Jannotti, J., Couto, D.S.J.D., Karger, D.R., Morris, R.: A scalable location service for geographical ad hoc routing. In: Proceedings of the ACM MOBICOM'00, pp. 120–130. Boston, MA (2000)
22. Li, X., Kim, Y.J., Govidan, R., Hong, W.: Multidimensional range queries in sensor networks. In: Proceedings of the ACM SenSys'03, pp. 63–75. Los Angeles, CA (2003)

23. Litwin, W., Risch, T.: LH*g: A high-availability scalable distributed data structure by record grouping. IEEE Transactions on Knowledge and Data Engineering **14**(4), 923–927 (2002)
24. Misra, A., Castro, P., Lee, J.: CLASH: A protocol for Internet-scale utility-oriented distributed computing. In: Proceedings of the International Conference on Distributed Computing Systems, pp. 273–281. Tokyo, Japan (2004)
25. Mondal, A., Yilifu, Kitsuregawa, M.: P2PR-tree: An R-tree-based spatial index for peer-to-peer environments. In: Proceedings of the International Workshop on Peer-to-Peer Computing and Databases (held in conjunction with EDBT), pp. 516–525. Heraklion-Crete, Greece (2004)
26. Ramabhadran, S., Ratnasamy, S., Hellerstein, J.M., Shenker, S.: Prefix hash tree. In: Proceedings of ACM PODC, p. 368. St. Johns, Canada (2004)
27. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: Proceedings of the ACM SIGCOMM'01, pp. 161–172. San Diego, CA (2001)
28. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proceedings of the ACM Middleware'01, pp. 329–350. Heidelberg, Germany (2001)
29. Sahin, O.D., Gupta, A., Agrawal, D., El Abbadi, A.: A peer-to-peer framework for caching range queries. In: Proceedings of the 20th IEEE International Conference on Data Engineering, pp. 165–176. Boston, MA (2004)
30. Samet, H.: Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS. Addison-Wesley, Reading, MA (1990)
31. Samet, H.: The Design and Analysis of Spatial Data Structures. Addison-Wesley, Reading, MA (1990)
32. Samet, H.: Foundations of Multidimensional Data Structures. Morgan Kaufmann, San Francisco (2005)
33. Sevcik, K., Koudas, N.: Filter trees for managing spatial data over a range of size granularities. In: Proceedings of the International Conference on Very Large Databases-VLDB, pp. 16–27. Mumbai, India (1996)
34. Silaghi, B., Bhattacharjee, B., Keleher, P.: Query routing in the TerraDir distributed directory. In: Proceedings of the SPIE ITCOM'02. Boston, MA (2002)
35. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for Internet applications. In: Proceedings of the ACM SIGCOMM'01, pp. 149–160. San Diego, CA (2001)
36. Tanin, E., Harwood, A., Samet, H., Nutanong, S., Truong, M.: A serverless 3D world. In: Proceedings of the Symposium on Advances in Geographic Information Systems, pp. 157–165. Arlington, VA (2004)
37. Ulrich, T.: Loose octrees. In: M. DeLoura (ed.) Game Programming Gems, pp. 444–453. Charles River Media, Rockland, MA (2000)
38. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.: Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications **22**(1), 41–53 (2004)