DPTree: A Balanced Tree Based Indexing Framework for Peer-to-Peer Systems

Mei Li Wang-Chien Lee Anand Sivasubramaniam
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802
E-Mail: {meli, wlee, anand}@cse.psu.edu

***Abstract*— Peer-to-peer (P2P) systems have been widely used for exchange of voluminous information and resources among thousands or even millions of users. Since shared data are normally identified by multiple attributes, a fundamental issue in P2P systems is to efficiently support complex queries on multi-dimensional data. Prior works suffer from some fundamental limitations, such as being constrained to support certain types of queries, excessive maintenance overheads, and etc. In this study, we propose a framework, called *distributed peer tree (DPTree)*, which efficiently supports various types of queries on multi-dimensional data in P2P systems based on balanced tree indexes. DPTree achieves the efficiency through the following designs: 1) distributing the tree structure among peers in a way preserving the nice properties of balanced tree structures yet avoiding single points of failure and performance bottlenecks; 2) organizing peers into an overlay structure that enables efficient navigation yet is easy to maintain; 3) an efficient navigation algorithm; 4) an innovative wavelet-based load balancing mechanism. Through extensive performance evaluation, we verify the superiority of DPTree over existing works.**

I. Introduction

Peer-to-peer (P2P) systems have been widely used for exchange of voluminous information and resources among thousands or even millions of users. Data objects shared in P2P systems, such as images, and documents, are normally identified by multiple attributes, and can be viewed as points in a multi-dimensional space. Users often issue complex queries in addition to point queries (exact match queries) to retrieve data objects from P2P systems. For instance, a user might issue following queries: ”return all documents with similarity to *x* within r” (range query), or ”return the *k* documents most similar to *x*” (K nearest neighbor query). Thus, one of the fundamental issues faced by P2P systems is to efficiently support complex queries (in addition to point query) on multi-dimensional data objects. Two most common types of complex queries are range query and K nearest neighbor (KNN) query. Given a reference data object q and a radius r, a range query returns the data objects whose difference from q is less than r. Given a reference data object q and an integer K, KNN query returns the K data objects most similar to q.

Although some techniques have been proposed to address query processing in P2P systems, they have some fundamental limitations. Distributed hash tables (DHTs), e.g., CAN [15] and Chord [16], can only support point queries based on identifiers. Other works propose techniques based on locality-preserving hashing (e.g., [2]), multiple index structures (e.g., [3]), or data space partitioning (e.g., [4], [19]). These works suffer from one or more of the following limitations: being constrained to support certain types of queries, excessive maintenance overheads, performance degradation under skewed data distribution, and ignorance of nonuniform accesses to different data objects.

In this study, we propose an indexing framework, called *distributed peer tree (DPTree)*, which efficiently supports various types of queries on multi-dimensional data in P2P systems at reasonable maintenance overheads. In addition, DPTree automatically adapts to data distribution and access pattern. DPTree is inspired by balanced tree indexes (R-tree [6] and a series of variants), which have been intensively studied and become well-accepted multi-dimensional index structures in database community over the years. They possess some nice features including the ability to efficiently support a rich set of queries and adaptivity to data distribution. However, it is challenging to support balanced tree indexes in P2P systems. Simply mapping each tree node to a peer would result in performance bottlenecks and single points of failure at the peers taking charge of the tree nodes at higher level. In addition, coupling a tree node with a peer (and coupling the tree data structure with the overlay structure) would make the maintenance complex and costly. DPTree overcomes this challenge by decoupling a tree node from a peer (and decoupling the tree data structure from the overlay structure) and assigning (replicating) tree nodes to peers in accordance with their access frequencies.

While the above basic idea of DPTree is straightforward, the following issues need to be addressed carefully. 1) *Tree distribution*: how tree nodes are assigned (replicated) among peers in accordance with their access frequencies. This is crucial to the search performance and maintenance overheads. 2) *Overlay structure*: how peers form into an overlay that facilitates efficient navigation on the tree yet is easy to maintain. 3) *Navigation*: how a query request is propagated to the destination. With each peer only having a partial view of the tree structure, navigation in DPTree is nontrivial. 4) *Access load balancing*: how the access load is fairly distributed among peers. Majority of existing works attempt to achieve fair distribution of storage load in the system through random hashing. However, in order to efficiently support complex queries, data objects in DPTree are organized in accordance with their attribute values rather than randomly hashed values.

In addition, accesses to different data objects are not uniformly distributed, and fair distribution of storage load does not imply fair distribution of access load. Therefore, we need to design an explicit access load balancing mechanism in DPTree.

To address these issues, we propose a suite of efficient solutions, which constitute the following four major contributions of this paper. 1) We propose *tree branch oriented distribution*, which distributes (and replicates) *tree branches*, i.e., the tree nodes along the path from the root to leaf nodes, as atomic units to different peers. This scheme correlates the number of replicas for a tree node to its access frequency, and thus it does not incur single points of failure and performance bottlenecks. 2) We propose a *tree-aware overlay*, which maps peers to an ID space in accordance with their assigned tree nodes, and then organizes peers into an overlay based on the design principle of skip graph [1] in the ID space. This overlay structure enables efficient tree navigation yet is easy to maintain. 3) We propose *aggressive navigation* algorithm, through which peers aggressively forward messages to the destinations in $log(N)$ steps on DPTree with high probability. 4) We propose *wavelet-based load balancing* mechanism, which leverages wavelet to monitor the load distribution in the system and adjust the load among peers in a light-weighted yet effective fashion.

As a proof of concept, we show how different types of queries, i.e., point query, range query and KNN query, can be easily and efficiently supported in DPTree. Through extensive performance evaluation, we demonstrate the superiority of DPTree over existing works on various aspects.

The rest of this paper is organized as follows. We provide the background and system model in next section. The design details of DPTree are given in Section III, and the algorithms to process different queries are described in Section IV. The performance evaluation is presented in Section V. We review the related studies in Section VI. Lastly, we draw the conclusion and outline future research in Section VII.

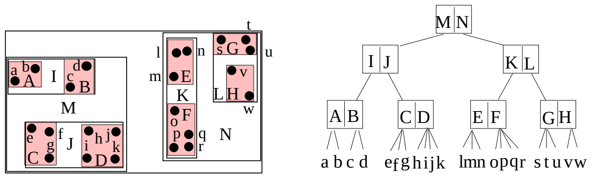
II. Preliminaries

A. Background

We briefly introduce balanced tree indexes, skip graph, and wavelet, which are necessary to understand DPTree.

1) Balanced Tree Index: Balanced tree indexes are hierarchical structures that recursively decompose a set of data objects into f (called *fanout*) subgroups (tree nodes). The decomposition stops when the number of data objects in a subgroup falls below a threshold value c (called *leaf node capacity*). The top level (coarsest) subgroup is the root node and the finest subgroups are the leaf nodes (other subgroups are called *non-leaf nodes*). A leaf node stores the *coverage* (enclosing region) of its corresponding subgroup and the storage addresses of the data objects in the subgroup. A non-leaf node stores the coverage of itself and its immediate children.

We use R-tree [6], a well-known balanced tree index, as an illustrative example. The coverage of a tree node in the R-tree is represented by the smallest rectangle enclosing all data objects in the subgroup, called *minimum bounding rectangle (MBR)*. Figure 1 shows an example. The left side depicts the MBRs and right side depicts the tree structure. Other balanced tree indexes have similar structures with different representation for the coverage of tree nodes.



a b A I c d B M e f g C J h i j k D l m n E K o F p q r N t s G u v L H w
M N
I J
K L
A B
C D
E F
G H
a b c d
e f g h i j k
l m n o p q r
s t u v w

Fig. 1. **An illustrative example of R-tree.**

To process a query in these index structures, the coverage of a tree node is examined against the query starting from the root node. If the coverage of a tree node does not enclose the query, the subtree rooted at this tree node does not need to be examined, i.e, the subtree can be *pruned*. Otherwise, the children's coverage is examined, and the query is continued at the child (or children) whose coverage encloses the query. The process continues till every part of the tree is either pruned or examined.

Balanced tree indexes have the following nice features. 1) Different from hashing techniques, they preserve locality among data objects, which is essential for efficient processing of complex queries. 2) They are fully dynamic and automatically adapt to data distribution. 3) They do not need to index *dead space* (the data space that is not populated by data objects), which is required in other overlays (e.g., CAN). 4) Mature algorithms for various types of queries based on these index structures have emerged over the years.

2) Skip Graph: Different from other overlays that establish overlay links based on the distance in an ID space, skip graph (or skipnet) establishes overlay links based on *peer distance* (the number of peers between two peers) [1], [7]. Therefore, skip graph is insensitive to the distribution of peers in the ID space. In skip graph, peers form into a doubly-linked ring. In addition, a peer maintains $(logN\text{-}1)$ skip pointers (neighbors), each of which skips over 2^i peers with high probability $(1 \leq i \leq logN\text{-}1)$. Conceptually, skip graph is a hierarchical ring with $logN$ levels. The level-0 ring consists of all peers. It is split into two *child rings* (the ring before splitting is called *parent ring* accordingly), which are then recursively split until the number of peers in the ring is not greater than 2. A peer joins one ring at each level.

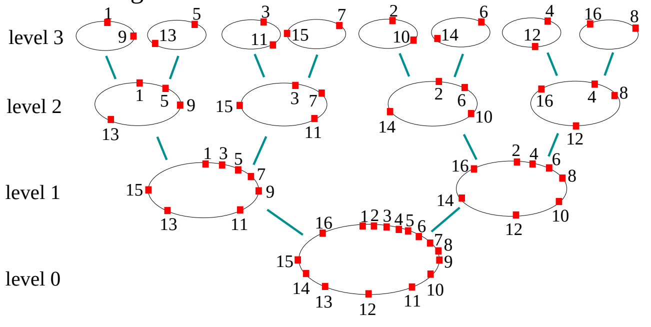


Fig. 2. **An illustrative example of skip graph.**

We first explain how to form a "perfect" skip graph, where a peer's neighbor at level-i of the overlay is at exactly 2^i peer distance away. Starting from level-0, the peers in a parent ring are alternatively assigned to one of the two child rings. Figure 2 illustrates an example of a four-level skip graph formed by 16 peers. Peer 1, 3, 5, 7, 9, 11, 13, and 15 form one child ring at level-1 and the rest of the peers form the other child ring at level-1. Each peer has four sets of neighbors with one set at each level.

The above overlay construction is too rigid to accommodate dynamic peer join/leave/failure. Therefore, some randomness is introduced to make the overlay flexible by allowing a peer to randomly join one of the two child rings. With high probability, a peer's neighbor at level-i is at 2^i peer distance.

Routing is performed level-by-level starting from the top level. At a specific level, the routing message is always forwarded to the neighbor that is closest to the destination without overshooting it. If no such neighbor can be found, routing descends one level and the above process repeats. It is proven that routing can be resolved in $logN$ steps with high probability in skip graph [1].

3) Wavelet: Wavelet is a tool used extensively in signal processing (for details, please see [17]). It provides views of data at different resolutions, called as *level of decomposition*. In the following, we introduce the basic concept of Harr Wavelet, which is simple and fast to compute. Harr Wavelet consists of *average coefficient* (or *average*) and *detail coefficients* (or *differences*) of a signal. The average coefficients and detail coefficients at a level of decomposition are obtained by pairwise averaging and differing of the averages on the previous level of decomposition.

We use a simple sequence consisting of {2, 4, 3, 3, 7, 13, 15, 1} to illustrate how Harr Wavelet is formed. The averages and differences at level-1 decomposition are obtained as {3, 3, 10, 8} and {2, 0, 6, -14}, respectively. We then repeat this process on the averages ({3, 3, 10, 8}) to get the averages and differences at level-2 decomposition as {3, 9} and {0, -2}, respectively. The level-3 decomposition is obtained similarly as {6} and {6}. The *wavelet transform* is defined as the average coefficient at the top level of decomposition, followed by the detail coefficients at increasing resolution (decreasing levels of decomposition). Thus the wavelet transform for the original 8-value signal is {6, 6, 0, -2, 2, 0, 6, -14}. Each of the 8 individual coefficients is called *wavelet coefficient*. In the remaining discussion, we refer wavelet transform as wavelet.

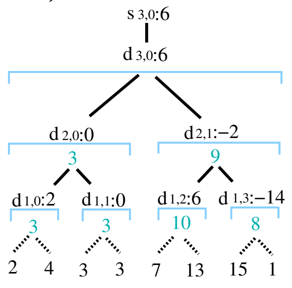


Fig. 3. **An illustrative example of wavelet error tree.**

Denoting the k^{th} average coefficient and detail coefficient at level-j decomposition as $s_{j,k}$ and $d_{j,k}$, respectively, the wavelet can be represented by a binary tree structure, called as *error tree* in the literature. In the error tree, the average at the top level of decomposition is the root of the tree. The detail coefficient at the top level of decomposition is the only child of the root. Each of the node representing a detail coefficient ($d_{j,k}$) has two children representing the detail coefficients at the next lower level of decomposition ($d_{j-1,2k}$ and $d_{j-1,2k+1}$). Figure 3 illustrates the error tree for the above 8-value signal. For illustration, we put the original values as the leaf nodes of the tree (depicted by dotted line). In addition, we depict the average coefficients for the levels other than the top level as lightly shaded numbers (they are not included in the wavelet transform/error tree).

The original signal can be reconstructed exactly from the wavelet coefficients by taking the reverse steps of decomposition. One desirable feature of wavelet transform is that many detail coefficients turn out to be very small and setting them to 0 introduces only small errors in the signal reconstruction. Therefore, the original signal can be approximated by a small number of the most significant wavelet coefficients.

B. System Model

Without loss of generality, we consider balanced binary tree (fanout $f = 2$), i.e., a binary tree where the height of the left subtree and right subtree of any tree node differs by at most 1. Note that our proposal is not limited to fanout of 2 though. The P2P system consists of N peers with homogeneous resources[1]. We call the peer owning a data object as this data object's *owner peer*, and the peer storing the index of a data object as this data object's *index peer*. The index of a data object is a tuple $\langle$value vector, location$\rangle$ where the value vector is a vector of values of the data object on different attributes, and the location is the identifier (IP address) of the owner peer.

III. DPTree

We first present the tree branch oriented distribution and the high level features of DPTree in Section III-A. We then discuss the overlay structure and navigation algorithm in Section III-B. The access load balancing mechanism is presented in Section III-C. Lastly, we explain how to maintain the overlay structure and tree structure upon peer join/leave/failure and data insertion/deletion in Section III-D.

A. Overview of DPTree

We explain the design rationale for tree branch oriented distribution before presenting the details of this scheme.

First, as mentioned earlier, coupling each tree node with a peer would incur performance bottlenecks and single points of failure at the peers responsible for the higher levels of the tree. Thus, it is better to decouple the concept of a tree node from a peer. This not only avoids the aforementioned problems, but also renders the system the flexility to develop/optimize individual mechanisms for tree maintenance, overlay maintenance, and load balancing according to the patterns of data update, peer update and load distribution, respectively.

Second, we observe that access on a tree normally proceeds from the root down to a leaf node of interest by traversing all non-leaf nodes along the path. This observation implies that the tree nodes on a tree branch (consisting of the tree nodes along the path from the root to a leaf node) are accessed together. Thus, it is beneficial to distribute the tree nodes on a tree branch to the same (or same set of) peer(s).

Based on these observations, we propose tree branch oriented distribution, which distributes (and replicates) tree branches as atomic units to different peers. Each peer manages

[1]In the case when peers have heterogeneous resources, we can use proper weighting functions as suggested in [14].

one or more tree branches (leading to one or more leaf nodes), which form the *local tree* of this peer. A peer stores the index of the data objects enclosed in the assigned leaf nodes. Thus, this peer is the index peer of these data objects. In addition, for each non-leaf node in the local tree, a peer stores the coverage and height of its two immediate child nodes, used for tree navigation and tree balance invariance checking, respectively. Note that the two immediate children of a non-leaf node in the local tree may or may not be present in the local tree, called *local child node* and *remote child node*, respectively.

Figure 4 illustrates an example for DPTree. The leaf nodes are grouped into 16 partitions (depicted by the grey rectangles), which are allocated to 16 peers. For the clarity of presentation, we number the tree nodes as Node 1 to Node 53 and the partitions as P1 to P16. Each peer manages the assigned leaf nodes as well as the non-leaf nodes on the tree branches leading to the assigned leaf nodes. For readability, we only show the local trees of Peer 1, 11 and 16 depicted by the dotted polygons in the figure. For instance, the local tree of peer 16 consists of the assigned leaf nodes 19-22 and the non-leaf nodes on the tree branches leading to these leaf nodes, i.e., Nodes 37–38, 46, 50, 52, 53. Node 45, 49 and 51 are the remote child nodes of Node 50, 52, and 53 in the local tree of Peer 16, respectively.

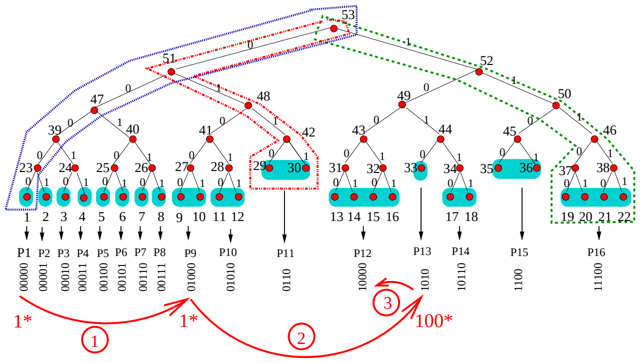


Fig. 4. **An illustrative example of DPTree.**

DPTree essentially is a fully distributed multi-dimensional balanced tree index. It inherits the nice properties of centralized balanced tree structures, i.e., locality preserving, adaptivity to dynamic data distribution, avoiding indexing dead space, and ability to support various types of queries. In addition, DPTree has its own unique feature, i.e., fair load distribution both vertically and horizontally. Vertically, the tree nodes at the higher levels of the tree have more replicas compared to the tree nodes at the lower levels of the tree. This design avoids the single points of failure and performance bottlenecks, which would normally be associated with the higher levels of a tree structure. Horizontally, the density of peers managing different portion of the tree (the number of tree branches assigned to different peers) reflects the access frequency of different data objects (through load balancing to be discussed later), i.e., more peers manage the frequently accessed portion of the tree.

B. *Overlay Structure and Navigation Algorithm*

1) Tree-Aware Overlay: One possible solution to construct the overlay is to couple the tree data structure with the overlay structure. That is, each peer maintains a pointer pointing to the other side of the subtree for each level of its local tree (i.e., the remote child node). However different peers might maintain different number of tree branches, resulting in different peers having different number of routing pointers. This would lead to uneven distribution of routing load among peers.

We propose to organize peers into a *tree-aware overlay*, which is decoupled from but aware of the tree data structure. In tree-aware overlay, peers first obtain a total order through *tree-aware peer naming scheme* and then form an overlay over the ordered space based on the design principle of skip graph (Section II-A.2). Tree-aware overlay enables efficient navigation on the tree yet is easy to maintain.

Tree-Aware Peer Naming. Tree-aware peer naming scheme assigns each peer a unique identifier, i.e., *peerID*. These peerIDs provide a total order of peers, which reflects the locality among assigned tree nodes. The naming scheme works as follows. Each edge in the tree is labelled as 0 or 1. A tree node obtains a *treenodeID*, which is the concatenation of the labels along the edges on the path from the root to this tree node. The treenodeID of a leaf node is called *leafID* specifically. While a peer might manage a couple of tree branches (and leaf nodes), it obtains its *peerID* as the smallest leafID among all the leafIDs of the leaf nodes that it manages. The lexicographic order among peerIDs defines the total order of the corresponding peers.

Figure 4 illustrates the naming scheme. The left edge and right edge of each node are labelled with 0 and 1, respectively. The peerIDs are depicted at the bottom of the figure[2]. For instance, Peer 11 manages two leaf nodes 29-30, which have leafID as 0110 and 0111, respectively. Thus, Peer 11 obtains its peerID as 0110. The left-to-right order depicted in the figure represents the total order of the 16 peers.

Skip Graph Based Overlay. Above naming scheme maps peers to an ID space. The distribution of peers in the ID space might not be uniform as a result of load balancing. Therefore, we need to construct an overlay that is insensitive to the skewed distribution of peers in the ID space. We observe that skip graph satisfies this requirement. Thus, we organize peers into a skip graph in the ID space. In addition, a peer periodically exchanges heartbeat messages with its neighbors to maintain the consistency of the overlay structure (to be detailed in Section III-D).

2) Aggressive Navigation: Navigation addresses how to navigate to the index peer managing the index of a requested data object[3]. Since each peer has only a partial view of the tree structure, navigation in DPTree is nontrivial. We first need to determine which part of the overlay (tree) might cover the requested data object or destination, i.e., *search space resolution*, and then get to that part of the overlay, i.e., *routing*.

Search Space Resolution. Since tree-aware overlay is constructed over the ID space, search space resolution is to estimate the leafID(s) of the leaf node(s) covering the destination, denoted as *destID*. This is achieved by examining a peer's local tree as follows. A peer examines the tree node

[2]treenodeID, leafID or peerID bear no relationship with the numbers used to label each tree node (1–53) and partition (P1–P16). The latter is for the purpose of illustration only.

[3]Once the index peer is found, the owner peer can be obtained easily from the index peer. Thus we focus on navigating to the index peer in this paper.

in its local tree with the treenodeID as the currently obtained destID (initially set to wildcard $*$). If there is no such tree node in its local tree, this peer can not refine the destID further. Otherwise, the child node that covers the destination is entered and examined further. This process continues till either a leaf node or a remote child node of the local tree is reached. In the former case, the current peer is the destination and the navigation terminates. In the latter case, destID is refined as $A*$, which indicates A, the treenodeID of this remote child node, is a prefix of the destID.

Routing. The destID (with wildcard $*$) obtained above through search space resolution actually represents a range of IDs in the ID space. Therefore, the routing algorithm for skip graph as mentioned in Section II-A.2 can not be simply applied here. Instead, we first need to decide which one of the IDs in the range specified by the destID should be used for routing. After careful examination, we observe that the ID furthest away from current peer's peerID should be used for routing. The rationale behind this aggressive navigation algorithm is that even though we might overshoot the destination by routing towards the furthest possible destID, the furthest possible distance to the destination is halved at each step. With the initial furthest possible distance to the destination as N, the navigation to the destination is finished in $logN$ steps with high probability (Theorem 1). Algorithm 1 illustrates the pseudo-codes for the navigation.

Algorithm 1 Algorithm for Aggressive Navigation in DPTree.

Navigation at Peer *i*: $i.navigate(q, destID, j)$ (destID indicates the estimated leafID for q. j, initialized to the top level of the overlay, indicates at which level of the overlay the routing should proceed.)

```
1: if $q \in i.index$ then
2:   Stop.
3: else
4:   Refine destID for $q$ by invoking search space resolution.
5:   $x$ = the furthest ID from Peer $i$ specified by destID.
6:   $m$ = level-$j$ neighbor of $i$ that is closest to $x$ without overshooting $x$.
7:   if m= NULL then
8:     j = j-1.
9:     GOTO 6.
10:  end if
11:  Forward $navigate(q, destID, j)$ to $m$.
12: end if
```

Theorem 1: Given N peers in DPTree, a peer can navigate to any part of the overlay in $O(logN)$ hops (steps) with high probability.

Proof: We prove this theorem by induction. Given the peer distance to the destination at the beginning of the i^{th} step as $x_i \leq \frac{N}{2^{i-1}}$, we prove the peer distance is halved after this routing step, i.e., $x'_i \leq \frac{N}{2^i}$. Once this is proven, we can easily derive after $logN$ steps, the peer distance to the destination is reduced to 1. The message can then be trivially forwarded to the destination with one additional step.

Assume the highest level of the overlay is h (h = $logN$ with high probability). Recall that each peer has $logN$ neighbors with the level-j neighbor at peer distance (denoted as d_j) as 2^j with high probability ($0 \leq j \leq logN$-1).

We start from the base case with $i = 1$. It is obvious that $x_1 \leq \frac{N}{2^{1-1}} = N$.

Assume that we are at the beginning of the i^{th} step and the peer distance to the destination is $x_i \leq \frac{N}{2^{i-1}}$. We have two possible scenarios: 1) $x_i > \frac{N}{2^i}$; 2) $x_i \leq \frac{N}{2^i}$. For the first scenario, one of the current peer's neighbors at level-(h-i) must be closer to the furthest possible destID without overshooting it. Therefore, this neighbor is chosen to forward the request. As a result, the peer distance to the destination (x'_i) is then reduced to $x_i - d_{h-i} = x_i - \frac{N}{2^i} \leq \frac{N}{2^i}$. For the second scenario, the request may or may not be forwarded at this routing step. If the request is forwarded, x'_i is reduced to $d_{h-i} - x_i$ ($\leq \frac{N}{2^i}$). Otherwise, x'_i equals to x_i ($\leq \frac{N}{2^i}$). $\square$

Let's go back to Figure 4 to illustrate an example for navigation in DPTree. Assume Peer 1 wants to navigate to leaf node 13 (please refer to Figure 2 for the overlay structure). Peer 1 invokes search space resolution and obtains the destID as $1*$. Since the peerID of Peer 1 is 00000, the furthest possible destID is 111...111. Peer 1 examines its neighbors at the top level and routes the message to Peer 9. When Peer 9 receives the message, it invokes search space resolution and still obtains the destID as $1*$. Similarly, Peer 9 examines its neighbors at level 3. Since the only neighbor at this level (Peer 1) is further away from the destID, routing descends one level to level 2. At this level, Peer 9 forwards the message to Peer 13, which is closer to $111...111$ without overshooting it. When Peer 13 receives the message, it invokes search space resolution and refines the destID as $100*$. The furthest destID from Peer 13 is 1000...000. Peer 13 examines its neighbors at level-2 and does not forward the message at this level since neither neighbor qualifies. Routing descends to level-1. Similarly, neither neighbor at level-1 qualifies and routing descends one more level to level-0, where the message is forwarded to the destination Peer 12.

C. Wavelet-assisted Load balancing

In the following, we first describe an intuitive solution for load balancing and point out the limitations, which motivate wavelet-assisted load balancing. We refer access load as load if the context is clear.

One solution for load balancing as suggested by majority of existing works (e.g., [3], [5], [10], [14]) is to let an overloaded peer choose the least loaded peer in the system as the *target peer* to shed part of its load to. The target peer merges its load to its neighbors, leaves its original place, and rejoins the overlay as the neighbor of the overloaded peer to take part of its load. We call this solution as *leave-rejoin* mechanism. This mechanism has two drawbacks. First, the peers in the neighborhood of the target peer might not be lightly loaded. Merging the load from the target peer to its neighbors might cause the neighbors become heavily loaded, resulting in cascading load balancing operations. Second, the leave-rejoin process causes changes on the overlay links, and thus requires update on the overlay, which might be costly. As suggested later, in some cases when the target peer and the overloaded peer are nearby in the overlay, it might be a better option to let the extra load ripple through the neighbors to reach the target peer without affecting the overlay structure.

We observe that if a peer somehow has a global view of the load distribution in the system, both weaknesses mentioned above can be avoided. To obtain such a global view, we need to summarize and disseminate the load distribution of the system in a compact yet sufficiently accurate format. Inspired by wavelet, a well-studied compression tool in signal procession, we propose wavelet-assisted load balancing, which leverages wavelet to facilitate *load monitoring* and *load adjusting* in P2P systems in a light-weighted yet effective fashion.

To perform load monitoring, peers exchange load information with their neighbors at each level of the overlay through heartbeat messages and form the approximate wavelet of the load distribution in the system, called *loadwavelet*. Load adjusting is performed in the following three steps. 1) Overloading monitoring: guided by loadwavelet, peers can easily determine whether they are overloaded. 2) Target peer selection: an overloaded peer leverages the multi-resolution view of loadwavelet to find a target peer, i.e., the peer that is lightly loaded and whose neighborhood is lightly loaded as well. 3) Load shedding: peers adjust their load[4] using two mechanisms, i.e., *rippled load shedding* and *direct load shedding*, depending on which one is more cost-effective by taking into consideration the cost incurred by index movement as well as overlay maintenance.

Compared to the aforementioned existing works, our load balancing mechanism has the following three advantages. 1) It uses a light-weighted mechanism to maintain a sufficiently accurate summary of the load distribution in the system. 2) When choosing a target peer, it takes into consideration the load on a peer as well as the load on the peers in its neighborhood. In contrast, prior works only consider the load on a peer itself. 3) It takes into account the cost incurred by both index movement and overlay maintenance, and switches between two different load shedding mechanisms depending on the cost. In contrast, prior works only consider the cost of index movement and conduct load shedding using a variant of direct load shedding.

We now explain the details of load monitoring and load shedding.

1) Load Monitoring: We first assume that we have a perfect skip graph (where a peer's neighbor at level i is at exactly 2^i peer distance) and unbounded communication resource to form an exact (complete) loadwavelet. Later on, we discuss how to relax these assumptions.

A peer first exchanges its current load with its level-0 neighbor clockwise on the overlay and forms the wavelet for the "signal" consisting of two values, i.e., its current load and its neighbor's current load. This peer then exchanges this wavelet with its level-1 neighbor clockwise on the overlay and forms the wavelet for the signal consisting of four values, i.e., the load on this peer and the following three consecutive peers. Following this process, through message exchange with a level-i neighbor on the overlay, the wavelet covering 2^{i+1} consecutive peers is obtained. This process continues till the top level of the overlay is reached. At this point, we obtain the loadwavelet of all the peers in the system.

We use Figure 5 to illustrate an example of loadwavelet in a system consisting of 16 peers (please refer to Figure 2 for the overlay structure). For illustration, we represent peers on each level of the overlay on a straight line. The numbers in the parentheses are the peers whose load values are included in the loadwavelet formed at the previous level of the overlay.

[4]We set the leaf node capacity c to be the payload size of a packet. The rationale is to let a leaf node also serve as the finest unit of load transferred between two peers during load balancing.

After message exchange with the neighbor at the top level (level-3), each peer obtains the loadwavelet of the system.

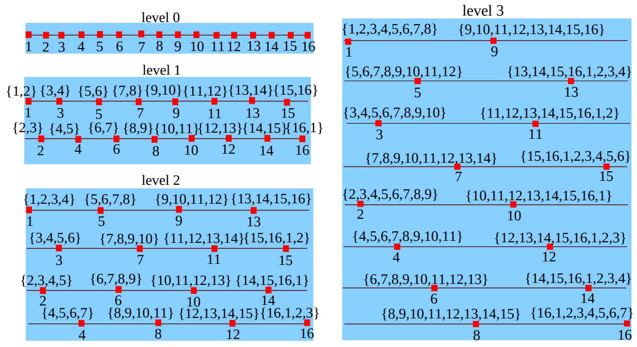


Fig. 5. **An illustrative example of loadwavelet.**

Now we discuss how to relax the assumption of unbounded communication resource. The length of the complete wavelet formed at higher levels of the overlay is large (e.g., the length of the loadwavelet formed at level-i of the overlay is 2^{i+1}), and exchanging such complete wavelets is costly. To reduce this cost, we select the m most significant wavelet coefficients to approximate the loadwavelet, where m is a tunable parameter balancing the precision and the construction cost of the wavelet (which will be evaluated in details later).

Up to now, we assume that a peer's level-i neighbor is at exactly 2^i peer distance. In reality, a peer's level-i neighbor is at 2^i peer distance with high probability instead of exactly. Therefore, during the above wavelet formation, the load values for certain peers might be counted more than once or not be counted (when a level-i neighbor is at peer distance less than 2^i or larger than 2^i, respectively). Since the percentage of redundant load values or missing load values is expected to be very small, this doesn't affect the accuracy of loadwavelet significantly.

2) Load Adjusting: As mentioned earlier, load adjusting consists of three tasks, i.e., overloading monitoring, target peer selection, and load shedding.

Overloading Monitoring. A peer obtains the average load of the system easily from the loadwavelet, i.e, the first wavelet coefficient. If the current peer's load is more than δ times of the average load, it is marked as an overloaded peer. δ is a tunable system parameter determining the tradeoff between the cost of load balancing and how well the system is balanced. A smaller value creates a more balanced system at higher cost.

Target Peer Selection. In addition to providing a compact summary of the load distribution, loadwavelet also provides multi-resolution views of the load distribution in the system. We exploit this multi-resolution feature to choose the lightly-loaded peer residing in a lightly-loaded neighborhood as the target peer. For presentation clarity, we use the wavelet error tree (described in Section II-A.3) to explain how target peer selection is performed. A peer examines the error tree starting from the root node. If the detail coefficient is greater than 0, the average load on the left half of the overlay is smaller. Thus, we drill down one level on the error tree and enter the left child. On the other hand, if the detail coefficient is smaller than 0, the right child is entered. In the case when the detail coefficient is 0, we examine both children's detail coefficients and enter the child node with larger absolute value (implying one quarter of the overlay has the lightest load among the four quarters). This process continues till we reach the bottom level of the error tree where the target peer is obtained. Using Figure 3 as an example for the load distribution in a system with 8 peers, target peer selection proceeds as follows. We first examine $d_{3,0}$. Since $d_{3,0}$ is a positive value, we enter the left child and examine $d_{2,0}$. $d_{2,0}$ is 0, and its left child node $d_{1,0}$ has larger absolute value. Thus, we enter the left child ($d_{1,0}$) and subsequently choose the first peer (with load as 2) as the target peer.

Load Shedding. Once a target peer is chosen, the load on the overloaded peer is adjusted through two different mechanisms, rippled load shedding and direct load shedding, depending on the cost incurred by both index movement and overlay maintenance. For presentation brevity, we describe the ideas of these two load shedding mechanisms. For the details on estimating the cost of these two mechanisms, please see [12].

In rippled load shedding, the load ripples through the neighbors to reach the target peer. The target peer first merges its load to its neighbors. Then the overloaded peer sheds half of its load to its immediate neighbor (by moving the corresponding leaf nodes), which then sheds the same amount of load to the neighbor's neighbor. This process continues till the target peer is reached. During rippled load shedding, although moving the leaf nodes between neighbors causes changes on the peerIDs of the peers involved, the order among peerIDs is not changed and thus the overlay links are not changed. Therefore, the advantage of rippled load shedding is that it incurs only index movement cost but no overlay maintenance cost. Rippled load shedding works well when the overloaded peer and target peer are nearby in the ID space.

Direct load shedding is similar to the load shedding scheme adopted in the leave-rejoin mechanism. It requires the target peer merge its load with its neighbors, leave its original place, and rejoin the overlay as a neighbor of the overloaded peer to take over half of its load. In addition to incurring index movement cost, the leave and rejoin of the target peer affects the overlay links, incurring overlay maintenance cost.

D. Maintenance in DPTree

We adopt the soft state mechanism to maintain the consistency of the overlay structure and tree data structure. The basic idea of soft state mechanism is to associate a state with a timer, and refreshes (or deletes) the state if a refreshment message is (or is not) received before the associated timer expires. Based on this idea, a peer associates a timer with each of its neighbors and each of its indexed data objects. A peer sends heartbeat messages to its neighbors periodically. If a peer does not receive a heartbeat message from a neighbor before the associated timer expires, it infers this neighbor leaves or fails (and invokes overlay update to be detailed shortly). Similarly, a peer republishes its data objects to the system periodically. If the index peer of a data object does not receive a refreshment message before the associated timer expires, it infers this data object disappears from the system (and invokes data deletion). This mechanism ensures the consistency of both overlay structure and tree data structure in a simple yet light-weighted fashion.

In the following, we provide the high level description on the operations performed upon peer join/leave/failure and data insertion/deletion. For more details, please see [12].

1) Peer Join/Leave/Failure: A peer joins the overlay level-by-level starting from level-0 by establishing two neighbors at each level. This process incurs $2logN$ messages in total. In addition, the newly joined peer publishes the index of the data brought with it to the system (to be detailed shortly).

Once a peer detects that one of its neighbors at level-i is not alive (due to lack of heartbeat messages as described above), it starts to re-establish its neighbors level-by-level starting from level-i. This process incurs at most $2logN$ messages in total. In addition, the tree branches previously assigned to this peer is re-distributed to other alive peers through the index republishing process as described above.

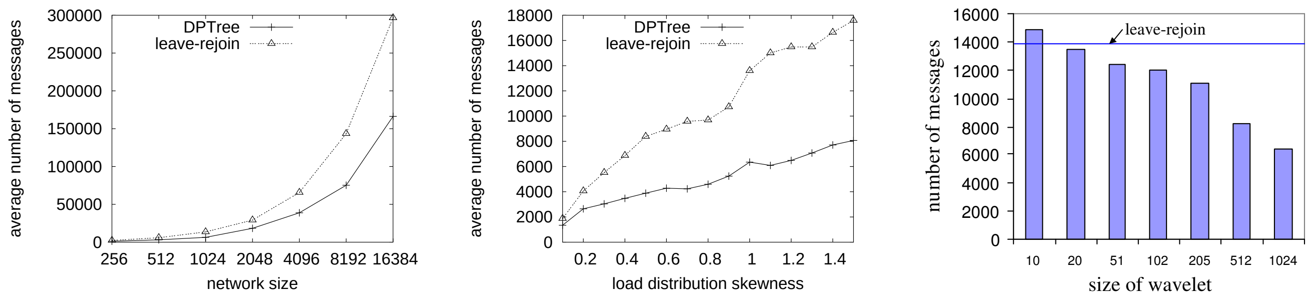
2) Data Insertion/Deletion: Inserting (deleting) a data object basically is to publish (remove) the index of this data object. This involves two steps: locating the leaf node (and corresponding peer) to insert (remove) the index through the similar procedure as tree navigation, and inserting (deleting) the index of the data object on the chosen leaf node (and splitting or merging the leaf node when necessary). The changes on the coverage and height of the tree nodes upon data insertion/deletion are propagated to the peers managing the tree nodes in the subtree rooted at the parent node of the tree node performing the changes (since all these peers record the coverage/height information of this node). We observe that through the tree aware peer naming scheme (Section III-B.1), all these peers are consecutively positioned in the ID space. Therefore, this update incurs n messages and $logn$ propagation hops where n is the number of affected peers. In addition, with the increase of the tree levels, the changes on the corresponding tree nodes become less frequent. This is beneficial since the propagation of the coverage/height changes on the tree nodes at the higher levels is more costly than that at the lower levels. Note that all these changes on the tree structure do not affect the overlay structure since the total order among peers does not change. This confirms the advantage of decoupling the tree structure from the overlay structure as discussed in Section III-B.1.

IV. Application of DPTree

In the following, we show how to support two most common types of complex queries, i.e., range query and KNN query, in DPTree (to support point query, we can directly apply the navigation algorithm presented in Section III-B.2 with the query as the destination).

A. Range Query

We extend the navigation algorithm given earlier (Section III-B.2) to process range query. The difference of a range query from a point query is that the destination is specified as a query range instead of a query point. Therefore, during search space resolution, all the tree nodes overlapping with the query range need to be entered and examined. If multiple child nodes need to be examined, multiple threads of processing are invoked to examine these nodes in parallel.



(a) network size (b) initial load distribution (c) wavelet size

Fig. 6. **Cost of load balancing.**

B. K Nearest Neighbor Query

A KNN query is processed in two steps, obtaining a good enough candidate set, and refining the candidate set through range query. To obtain a good enough candidate set, the peer managing the leaf node that covers the reference data object or that is closest to the reference data object (in the case that none of the leaf nodes covers the reference data object) is reached through the navigation algorithm. This peer then obtains the K data objects (either owned or indexed by this peer) that are closest to the reference data objects as the candidate set. It is possible that some data objects in other peers might be closer to the reference data object than the data objects in the candidate set are. In order to obtain these closer data object, the second step is invoked as follows. The current peer obtains a query range centered at the reference data object with the distance to the K^{th} element in the candidate set as the radius. Then similar procedure as range query is employed. As soon as a closer data object is obtained, the candidate set and the query range are refined. This process continues until the refined query range is completely examined.

V. Performance Evaluation

We now proceed to the evaluation of DPTree. We import R-tree according to the proposed DPTree framework. We evaluate DPTree from four aspects, i.e., routing, load balancing, query processing, and maintenance. For presentation brevity, we summarize the results for routing and maintenance and present the detailed results for load balancing and query processing (please see [12] for more details). We verify that routing to any part of the system is achieved within $logN$ steps in DPTree regardless of the distribution of peers in the ID space. In addition, we verify the average maintenance overhead incurred per peer join/leave/failure and data insertion/deletion is low, confirming our discussion in Section III-D.

A. Load Balancing

To evaluate the performance of our proposed load balancing mechanism, we measure the number of messages required to make the system δ-balanced, defined as a system with $\forall i \in \{0, 1, 2, ..., N\}, l_i \leq \delta \cdot \bar{l}$ where l_i is the load of Peer i and $\bar{l}$ is the average load of the system. For comparison, we implement the leave-rejoin mechanism (proposed in Mercury [3]) as described in Section III-C. In the leave-rejoin mechanism, the least loaded peer is chosen as the target peer and the load shedding process is similar to direct load shedding.

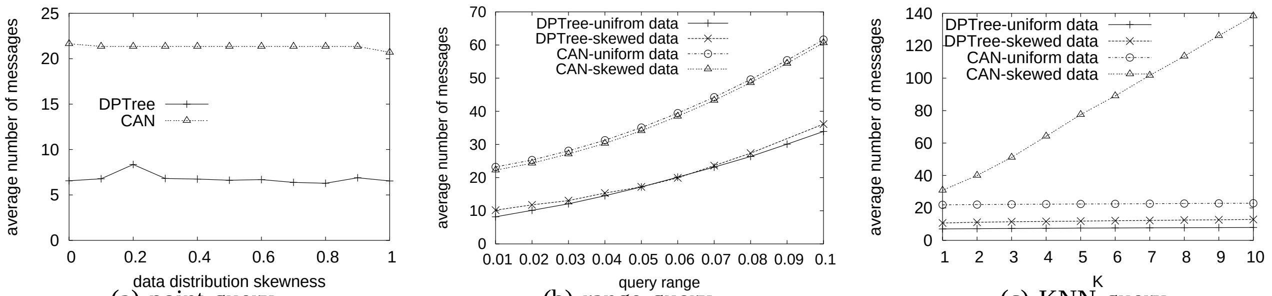
We evaluate the number of messages incurred by load balancing under different network sizes, different initial access load distribution, and different sizes of (approximate) wavelet (indicated by m as discussed in Section III-C.1). We use Zipf-distribution controlled by *load distribution skewness* to model the initial access load distribution. The default setting for the network size, load distribution skewness, and wavelet size is 1024, 1, and N, respectively. For presentation brevity, we present the results with δ set to 2 (the general trends observed under different setting for δ are similar).

1) Effect of Network Size: Figure 6(a) shows the result under different network sizes. The x-axis is on logarithmic scale for readability. From this figure, we see the number of messages incurred by load balancing increases almost linearly with the network size, which is expected. In addition, the number of messages incurred by wavelet-assisted load balancing is smaller than that incurred by the leave-rejoin mechanism, especially when the network size is large.

2) Effect of Initial Load Distribution: Figure 6(b) shows the result under different initial load distribution (different skewness values). The number of messages increases with the skewness value. This is expected since more skewed load distribution requires more load to be redistributed among peers to make the system δ-balanced. The increase rate of the cost incurred by wavelet-assisted load balancing is much smaller than that incurred by the leave-rejoin mechanism. This demonstrate the superiority of wavelet-assisted load balancing under more skewed load distribution.

3) Effect of Wavelet Size: Figure 6(c) shows the result under different wavelet sizes, i.e., 10, 20, 51, 102, 204, 512, corresponding to 1%, 2%, 5%, 10%, 20% and 50% of the size of the original wavelet transform (1024). From this figure, we see that when the wavelet size decreases, the number of messages increases. This is because more errors are introduced in signal reconstruction by a more compact approximate wavelet, which causes the selection of the target peers deviate from the optimal ones, incurring some extra messages. However, even when the size of the approximate wavelet is only 2% of the original wavelet, the number of messages incurred by wavelet-assisted load balancing is still smaller than that incurred by the leave-rejoin mechanism (indicated by the horizontal line in the figure).

This set of experiments demonstrates the superiority of wavelet-assisted load balancing. In addition, a compact approximate wavelet can be formed as the by-product of heart-beat messages exchange between neighbors at almost no ad-



(a) point query (b) range query (c) KNN query

Fig. 7. **Query performance.**

ditional cost. In contrast, the leave-rejoin mechanism requires non-trivial maintenance to keep track of the load distribution in the system (to be explained in Section VI).

B. Query Performance

We implement the algorithms to process three different queries, i.e., point query, range query and KNN query, in DPTree. For comparison, we modify CAN overlay (by placing data objects in accordance with their attribute values instead of randomly hashed values) and implement the query algorithms (similar to that described in Section IV) on top of CAN. We choose CAN overlay for comparison in this set of experiments due to the following two reasons. First, majority of prior works are based on CAN or some variants of CAN. Second, although some proposals (e.g., [3]) might be superior to CAN in terms of supporting one specific type of query, they can not support all types of queries that we are considering here.

We evaluate the query performance under different data sets, query workloads, and network sizes. Without loss of generality, the dimensionality of data objects is set to 2. The data sets are generated as follows. A certain number of seed points are first randomly generated in the 2-dimensional data space. Then from each of these seed points, we generate some random data points with distance to the seed point following Zipf distribution controlled by *data distribution skewness*. By varying the skewness value, we obtain a spectrum of synthetic data sets ranging from uniformly distributed data set to highly skewed data set. The total number of data points is $100 \cdot N$.

The query workload is generated as follows. We randomly select a point as the query point for point query or reference data object for range query and KNN query. For range queries, we vary the query radius from 0.01 to 0.1. For KNN queries, we vary the value of K from 1 to 10. We inject 1000 random queries into the system and the results presented below are the average results over these queries. Since the general trends observed under different network sizes are similar, we only show the results under network size as 1024.

1) Point Query: Figure 7(a) shows the result of point query with data distribution skewness varying from 0 to 1. The number of messages incurred by DPTree is always smaller than that of CAN. This confirms the efficiency of the tree-aware overlay and aggressive navigation algorithm, and the benefits of avoiding indexing dead space in DPTree as discussed in Section II-A.1. In addition, the number of messages incurred by DPTree is insensitive to data distribution skewness. This confirms the adaptivity of DPTree to data distribution. It seems the number of messages incurred by CAN also does not change significantly under different skewness values. However, this comes with the price of uneven load distribution among peers due to the naive data space partition scheme adopted in CAN. In contrast, DPTree always achieves fair load distribution through the initial load-aware data placement using branch-oriented tree distribution and the subsequent wavelet-based load balancing.

2) Range Query: Figure 7(b) shows the result of range query with query range varying from 0.01 to 0.1. For readability, we only present the result with data distribution skewness set to 0 and 1, respectively. As expected, the number of messages increases with the query range. In addition, the number of messages incurred by DPTree is always smaller than that incurred by CAN. This again confirms the efficiency of our tree-aware overlay and navigation algorithm, and the benefits of avoiding indexing dead space in DPTree.

3) KNN Query: Figure 7(c) shows the results of KNN query with the K value varying from 1 to 10. Similarly, we only present the result with data distribution skewness set to 0 and 1, respectively. From this figure, we see that the number of messages incurred by DPTree increases very slowly with the K value under both uniform data set and skewed data set, and the number of messages incurred by DPTree is always smaller than that of CAN. Furthermore, the number of messages incurred by DPTree under skewed data set is only slightly larger than that under uniform data set. In addition to validating the efficiency of our tree-aware overlay and navigation algorithm, this further confirms DPTree is adaptive to data distribution, which benefits KNN query processing. In contrast, the number of messages incurred by CAN under skewed data set increases rapidly with K values. This shows lack of the ability to adapt to data distribution seriously degrades the performance of CAN under skewed data set.

VI. Related Works

A few recent works propose to organize peers into a balanced tree structure to support complex queries in P2P systems, e.g., BATON [8], and VBI [9]. BATON simply assigns each tree node to a peer and then establishes a chord-like routing structure on each level of the tree. It only works for one-dimensional data. In addition, BATON creates skewed traffic distribution. While the peers responsible for the lowest level of the tree perform majority of the routing, the peers responsible for the higher levels of the tree are hardly used. VBI [9] extends BATON to support multi-dimensional data objects. However, it follows the design principle of BATON and also incurs skewed traffic distribution. Fat-Btree [18] is a distributed one-dimensional balanced tree (Btree) designed for parallel database system. As opposed to DPTree, all these works couple the tree data structure with the overlay, making the maintenance and load balancing complicated and costly.

A few works (e.g., [4], [19]) address complex queries on multi-dimensional data by proposing techniques based on data space partitioning. They have the following limitations. First, they do not adapt to the dynamic changes on data distribution. Second, as opposed to DPTree, these techniques have to index the whole data space including the dead space, increasing the index maintenance overhead as well as query cost. Although these works also construct tree structures, these tree structures are static and can become unbalanced, affecting the performance of these proposals in different aspects. SSW [11] is designed to facilitate efficient P2P search in high-dimensional space and addresses the research issues raised by high dimensionality in P2P systems.

Mercury [3] addresses multi-dimensional range query by constructing multiple index structures with one for each attribute. It incurs high index maintenance overheads, especially when the number of attributes is large. In addition, since different attributes are indexed separately, it is not clear how Mercury can be extended to support queries that require the interplay among different attributes, e.g., KNN query. Reference [2] proposes a technique based on locality preserving hashing, which mandates complex/costly index construction and only provides approximate query answers. NRtree [13] imports R-tree to a super-peer network, which is different from our focus here.

Quite a few works investigate load balancing. The studies that are most relevant to our work are [3], [5], [10], [14]. The central idea of these works is the leave-rejoin mechanism as described in Section III-C. These works differ in the way how a target peer is determined. [10], [14] rely on random communication with multiple peers, which can not guarantee that a chosen target is a lightly loaded peer and also incurs extra cost by randomly probing multiple peers. Reference [5] proposes to maintain a separate data structure recording the load on each peer, which is expected to be a nontrivial task given the large size and high dynamics of the system. Reference [3] relies on extensive sampling to maintain an approximate histogram of the load distribution in the system.

VII. Conclusion

One of the fundamental challenges faced by peer-to-peer (P2P) systems is to efficiently support complex queries on multi-dimensional data objects. Although some works have studied this issue, they suffer from some fundamental limitations. We propose a framework, called distributed peer tree (DPTree), to efficiently support various types of queries on multi-dimensional data in P2P systems based on balanced tree indexes. DPTree combines a number of innovative ideas to achieve the efficiency: *tree branch oriented distribution*, *tree-aware overlay*, *aggressive navigation*, and *wavelet-assisted load balancing*. We demonstrate the superiority of DPTree over existing works through extensive performance evaluation.

DPTree provides a sound foundation where various data management tasks can be explored. We plan to exploit DPTree to investigate other types of complex queries and various data mining tasks in P2P systems.

VIII. Acknowledgements

Wang-Chien Lee and Anand Sivasubramaniam were supported in part by National Science Foundation grant IIS-0534343. We would like to thank the anonymous reviewers for their constructive comments on the earlier version of this paper.

References

[1] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 384 – 393, January 2003.

[2] M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *Proceedings of International Conference on World Wide Web (WWW)*, pages 651–660, May 2005.

[3] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of ACM SIGCOMM*, pages 353–366, August 2004.

[4] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A case study in building layered DHT applications. In *Proceedings of ACM SIGCOMM*, pages 97–108, August 2005.

[5] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 444–455, August 2004.

[6] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD*, pages 47–54, 1984.

[7] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A scalable overlay network with practical locality properties. In *Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[8] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A balanced tree structure for peer-to-peer networks. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 661–672, September 2005.

[9] H. V. Jagadish, B. C. Ooi, Q. H. Vu, Z. Rong, and A. Zhou. VBI-Tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *Proceedings of Internatianal Conference on Data Engineering (ICDE)*, page to appear, April 2006.

[10] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of International Conference on Peer-to-Peer Systems (IPTPS)*, pages 131–140, February 2004.

[11] M. Li, W.-C. Lee, and A. Sivasubramaniam. Semantic small world: An overlay network for peer-to-peer search. In *Proceedings of International Conference on Network Protocols (ICNP)*, pages 228–238, October 2004.

[12] M. Li, W.-C. Lee, and A. Sivasubramaniam. DPTree: A balanced tree based indexing framework for peer-to-peer systems. Technical report, Pennsylvania State University, August 2006.

[13] B. Liu, W.-C. Lee, and D. L. Lee. Supporting complex multi-dimensional queries in P2P systems. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, pages 155–164, June 2005.

[14] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured P2P systems. In *Proceedgins of International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 68–79, February 2003.

[15] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172, August 2001.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM*, pages 149–160, August 2001.

[17] E. J. Stollnitz, T. D. DeRose, and D. H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann, 1996.

[18] H. Yokota, Y. Kanemasa, and J. Miyazaki. Fat-Btree: An update-conscious parallel directory structure. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 448–457, March 1999.

[19] C. Zhang, A. Krishnamurthy, and R. Y. Wang. Brushwood: Distributed trees in peer-to-peer systems. In *Proceedings of International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 47–57, February 2005.