MIDAS: Multi-Attribute Indexing for Distributed Architecture Systems

George Tsatsanifos[1], Dimitris Sacharidis[2⋆], and Timos Sellis[3]

[1] National Technical University of Athens, Greece,
gtsat@dblab.ece.ntua.gr
[2] Institute for the Management of Information Systems, R.C. "Athena", Greece,
dsachar@imis.athena-innovation.gr
[3] National Technical University of Athens, Greece, and
Institute for the Management of Information Systems, R.C. "Athena", Greece,
timos@imis.athena-innovation.gr

Abstract. This work presents a pure multidimensional, indexing infrastructure for large-scale decentralized networks that operate in extremely dynamic environments where peers join, leave and fail arbitrarily. We propose a new peer-to-peer variant implementing a virtual distributed k-d tree, and develop efficient algorithms for multidimensional point and range queries. Scalability is enhanced as each peer has only partial knowledge of the network. The most prominent feature of our method, is that in expectance each peer maintains $O(\log n)$ state and requests are resolved in $O(\log n)$ hops with respect to the overlay size n. In addition, we provide mechanisms for handling peer failures and improving fault tolerance as well as balancing the load of peers. Finally, our work is complemented by an experimental evaluation, where MIDAS is shown to outperform existing methods in spatial as well as in higher dimensional settings.

1 Introduction

Peer-to-peer (P2P) systems have emerged as a popular technique for exchanging information among a set of distributed machines. Recently, structured peer-to-peer systems gain momentum as a general means to decentralize various applications, such as lookup services, file systems, content delivery, etc. This work considers the case of a structured distributed storage and index scheme for multidimensional information, termed MIDAS, for **M**ulti-Attribute **I**ndexing for **D**istributed **A**rchitecture **S**ystems. The important feature of MIDAS is that it is capable of efficiently processing the most important types of multi-attribute queries, such as point and range queries, in arbitrary dimensionality.

While a lot of research has been devoted to structured peer-to-peer networks, only a few of them are capable of indexing multidimensional data. We distinguish three categories. The first includes solutions based on a single-dimensional P2P method. The most naive method is to select a single attribute and ignore all others for indexing, which clearly has its disadvantages. A more attractive alternative is to index each dimension

⋆ The author is supported by a Marie Curie Fellowship (IOF) within the European Community FP7.

separately, e.g., [4], [6]. However, these approaches still have to resort to only one of the dimensions for processing queries. The most popular approach, [9], [18], [5], within this category is to map the original space into a single dimension using a space filling curve, such as Hilbert or z-curve, and then employ any standard P2P system. These techniques suffer, especially in high dimensionality, as locality cannot be preserved. For instance, a rectangular range in the original space corresponds to multiple non-contiguous ranges in the mapped space.

The second category contains P2P systems that were explicitly designed to store multidimensional information, e.g., [15], [9]. The basic idea in these methods is that each peer is responsible for a rectangular region of the space and it has knowledge of its neighbors in adjacent regions. Being multidimensional in nature, allows them to feature sublinear to the network size cost for most queries. Their main weakness, however, is that they cannot take advantage of a hierarchical indexing structure. As a result, lookups for remote (in the multidimensional space) peers are unavoidably routed through many intermediate node, i.e., jumps cannot be made.

The last category includes methods, e.g., [10], [11], that decentralize a conventional hierarchical multidimensional index, such as the R-tree. The basic idea is that each peer corresponds to a node (internal or leaf) of the index, and establishes link to its parent, children and selected nodes at the same depth of the tree but in different subtrees. Queries are processed similar to the centralized approach, i.e., the index is traversed starting from the root. As a result, these methods inherit nice properties like logarithmic search cost, but face a serious limitation. Peers that correspond to nodes high in the tree can quickly become overloaded as query processing must pass through them. While this was a desirable property in centralized indices in order to minimize the number of I/O operations by maintaining these nodes in main memory, it is a limiting factor in distributed settings leading to bottlenecks. Moreover, this causes an imbalance in fault tolerance: a peer high in the tree that fails requires a significant amount of effort from the system to recover. Last but not least, R-trees are known to suffer in high dimensionality settings, which carries over to their decentralized counterparts. For example, the experiments in [11] showed that for dimensionality close to 20, this method was outperformed by the non-indexed approach of [15].

Motivated by these observations, MIDAS takes a different approach. First, it employs a hierarchical multidimensional index structure, the k-d tree. This has a series of benefits. Being a binary tree, it allows for simple and efficient routing, in a manner reminiscent of Plaxton's algorithm [14] for single dimensional tree-like structures. Unlike other multidimensional index techniques, e.g., [11], peers in MIDAS only correspond to leaf nodes of the k-d tree. This, alleviates bottlenecks and increases scalability as no single peer is burdened with routing multiple requests. Moreover, MIDAS is compatible with conventional techniques for load balancing and replication-based fault tolerance.

In summary, MIDAS is an efficient method for indexing multi-attribute data. We prove that in expectance point queries and range queries are performed in $O(\log n)$ hops; these bounds are smaller than non-indexed multidimensional P2P systems, e.g., $O(d\sqrt[d]{n})$ of [15]. A thorough experimental study on real spatial data as well as on synthetic data of varying dimensionality validates this claim.

The remainder of this paper is organized as follows. Section 2 compares MIDAS to related work. Section 3 describes our index scheme and basic operations including load balancing and fault tolerance mechanisms. Section 4 discusses multidimensional query processing. Section 5 presents an extensive experimental evaluation of all MIDAS' features. Section 6 concludes and summarizes our contributions.

2 Related Work

Structured peer-to-peer networks employ a globally consistent protocol to ensure that any peer can efficiently route a search to the peer that has the desired content, regardless of how rare it is or where it is located. Such a guarantee necessitates a structured overlay pattern. The most prominent class of approaches is *distributed hash tables* (DHTs). A DHT is a decentralized, distributed system that provides a lookup service similar to a hash-table. DHTs employ a consistent hashing variant [12] that is used to assign ownership of a (key, value) pair to a particular peer of an overlay network. Because of their structure, they offer certain guarantees when retrieving a key (e.g., worst-case logarithmic number of hops for lookups, i.e., point queries, with respect to network size). DHTs form a reliable infrastructure for building complex services, such as distributed file systems, content distribution systems, cooperative web caching, multicast, domain name services, etc.

Chord [19] uses a consistent hashing variant to associate unique (single-dimensional) identifiers with resources and peers. A key is assigned to the first peer whose identifier is equal to, or follows the key, in the identifier space. Each peer in Chord has $\log n$ state, i.e., number of neighbors, and resolves lookups in $\log n$ hops, where n is the size of the overlay network, i.e., the number of peers.

Another line of work involves tree-like structures, such as P-Grid [1], Kademlia [13], Tapestry [20] and Pastry [17]. Peer lookup in these systems is based on Plaxton's algorithm [14]. The main idea is to locate the neighbor whose identifier shares the longest common prefix with the requested (single-dimensional) key, and repeat this procedure recursively until the owner of the key is found. Lookups cost $O(\log n)$ hops and each peer has $O(\log n)$ state. MIDAS is similar to these works in that it has a tree-like structure with logarithmic number of neighbors at each peer, but differs in that it is able to perform multidimensional lookups in $O(\log n)$ hops.

We next discuss various structured peer-to-peer systems that natively index multi-attribute keys. In CAN [15], each peer is responsible for its *zone*, which is an axis-parallel orthogonal region of the d-dimensional space. Each peer holds information about a number of adjacent zones in the space, which results in $O(d)$ state. A d-dimensional key lookup is greedily routed towards the peer whose zone contains the key and costs $O(d\sqrt[d]{n})$ hops. Analogous results hold for MURK [9], where the space is a d-dimensional torus. The main concern with these approaches is that their cost (although sublinear to n) is considerate for large networks.

Several approaches, e.g., SCRAP [9], ZNet [18], employ a space filling curve to map the multidimensional space to a single dimension and then use a conventional system to index the resulting space. For instance, [5] uses the z-curve and P-Grid to support multi-attribute range queries. The problem with such methods is that the locality of the

original space cannot be preserved well, especially in high dimensionality. As a result a single range query is decomposed to multiple range queries in the mapped space, which increases the processing cost.

MAAN [6] extends Chord to support multidimensional range queries by mapping attribute values to the Chord identifier space via uniform locality preserving hashing. MAAN and Mercury [4] can support multi-attribute range queries through single-attribute query resolution. They do not feature pure multidimensional schemes, as they treat attributes independently. As a result, a range query is forwarded to the first value appearing in the range and then it is spread along neighboring peers exploiting the contiguity of the range. This procedure is very costly particularly in MAAN, which prunes the search space using only one dimension.

The VBI-tree [11] is a distributed framework based on balanced multidimensional tree structured overlays, e.g., R-tree. It provides an abstract tree structure on top of an overlay network that supports any kind of hierarchical tree indexing structures, i.e., when the region managed by a node covers those managed by its children. However, it was shown in [5] that for range queries the VBI-tree suffers in scalability in terms of throughput. Furthermore, it can cause unfairness as peers corresponding to nodes high in the tree are heavily hit.

3 MIDAS Architecture

This section presents the information stored in each peer and details the basic operations in the MIDAS overlay network. In particular, Section 3.1 introduces the distributed index structure, Section 3.2 discusses the information stored within each peer in MIDAS. Section 3.4, 3.3 and 3.5 elaborates on the actions taken when a peer departs, joins, and fails, respectively. Section 3.6 discusses load balancing and fault tolerance.

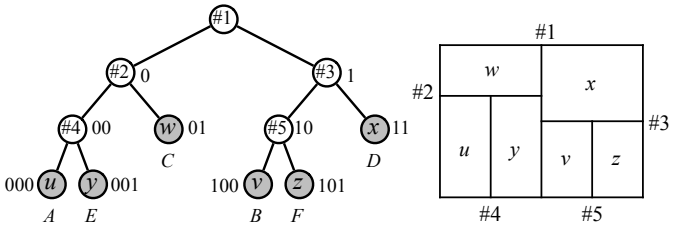
3.1 Index Structure

The distributed index of MIDAS is an instance of an adaptive k-d tree [3]. Consider a D-dimensional space $I = [\boldsymbol{\ell}_I, \boldsymbol{h}_I]$, defined by a low $\boldsymbol{\ell}_I$ and a high $\boldsymbol{h}_I$ D-dimensional point. The k-d tree T is a binary tree, in which each node $T[i]$ corresponds to an axis parallel (hyper-) rectangle I_i; the root $T[1]$ corresponds to the entire space, i.e., $I_1 = I$. Each internal node $T[i]$ has always two children, $T[2i]$ and $T[2i+1]$, whose rectangles are derived by splitting I_i at some value s_i along some dimension d_i; the splitting criterion (i.e., the values of s_i and d_i) are discussed in Section 3.3. Note that d_i represents the splitting dimension of node $T[i]$ and not the i-th dimension of the space.

Consider node $T[i]$'s two children, $T[2i]$, $T[2i+1]$, and their rectangles $I_{2i} = [\boldsymbol{\ell}_{2i}, \boldsymbol{h}_{2i}]$, $I_{2i+1} = [\boldsymbol{\ell}_{2i+1}, \boldsymbol{h}_{2i+1}]$. Assuming that the left child ($T[2i]$) is assigned the lower part of I_i, it holds that (1) $\boldsymbol{\ell}_{2i}[d_j] = \boldsymbol{\ell}_{2i+1}[d_j]$ and $\boldsymbol{h}_{2i}[d_j] = \boldsymbol{h}_{2i+1}[d_j]$ on every dimension $d_j \neq d_i$, and (2) $\boldsymbol{h}_{2i}[d_i] = \boldsymbol{\ell}_{2i+1}[d_i] = s_i$ on dimension d_i. We write $I_{2i} \uplus^{d_i} I_{2i+1}$ to denote that the above properties hold for the two rectangles.

Each node of the k-d tree is associated with a binary identifier corresponding to its path from the root, which is defined recursively. The root has the empty id $\varnothing$; the left (resp. right) child of an internal node has the id of its parent augmented with 0 (resp.

1). Figure 1a depicts a k-d tree of eleven nodes obtained from five splits; next to each node its id is shown. Due to the hierarchical splits, the rectangles of the leaf nodes in a k-d tree constitute a non-overlapping partition of the entire space I. Figure 1b draws the rectangles corresponding to the leaves of Figure 1a; the splits are numbered and shown next to the corresponding axis parallel cuts.



(a) A k-d tree (b) Leaf rectangles

Fig. 1: An example of a two-dimensional k-d tree.

A tuple with D attributes is represented as a point in the D-dimensional space I indexed by a k-d tree. A leaf of a k-d tree stores all tuples that fall in its rectangle. The hierarchical structure of the k-d tree allows for efficient methods to process queries, such as range queries, which retrieve all tuples within a range.

3.2 MIDAS Peers

It is important to distinguish the concepts of a physical and a virtual peer. A virtual peer, or simply a *peer*, is the basic entity in MIDAS. On the other hand, a *physical peer* is an actual machine that takes part in the distributed overlay. A physical peer can be responsible for several peers due to node departures or failures (Section 3.4, 3.5), or for load balancing and fault tolerance purposes (Section 3.6).

A peer in MIDAS corresponds to a leaf of the k-d tree, and stores/indexes all tuples that reside in the leaf's rectangle, which is called its *zone*. A peer is denoted with small letters, e.g., u, v, w, etc., whereas a physical peer with capital letters, e.g., A, B, C, etc. For example, in Figure 1a, physical peer C acts as the single peer w corresponding to leaf 01. We emphasize that internal k-d tree nodes, e.g., the non-shaded nodes in Figure 1a, do not correspond to peers and of course not to physical peers. An important property of peers in MIDAS is the following *invariant*.

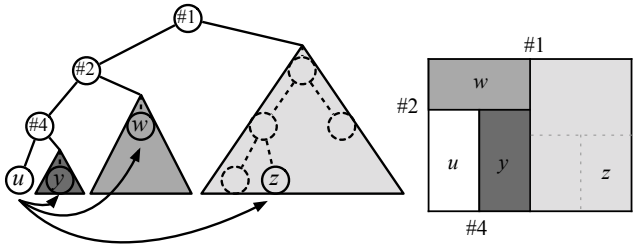
Lemma 1. *For any point in space I, there exists exactly one peer in MIDAS responsible for it.*

Proof. Each peer corresponds to a k-d tree leaf. The lemma holds because the leaves constitute an non-overlapping partition of the entire space I. □

A peer u in MIDAS contains only partial information about the k-d tree, which however is sufficient to perform complex query processing discussed in Section 4. In particular, peer u contains the following state. (1) $u.id$ is a bitmap representing the

leaf's binary id; $u.id[j]$ is the j-th most significant bit. (2) $u.depth$ is the depth of the leaf in the k-d tree, or equivalently the number of bits in $u.id$. (3) $u.sdim$ is an array of length $u.depth$ so that $u.sdim[j]$ is the splitting dimension of the parent of the j-th node on the path from the root to u. (4) $u.split$ is an array of length $u.depth$ so that $u.split[j]$ is the splitting value of the parent of the j-th node on the path from the root to u. (5) $u.link$ is an array of length $u.depth$ that corresponds to u's routing table, i.e., it contains the peers u has a link to. (6) $u.backlink$ is a list that contains all peers that have u in their $link$ array.

In the following, we explain the contents of $u.link$, which define the routing table of peer u. First, we define an important concept. Consider the prefixes of u's identifier; there are $u.depth$ of them. Each prefix corresponds to a subtree of the k-d tree that contains the leaf u (more accurately the leaf that has id $u.id$) and identifies a node on the path from the root to u. In the example of Figure 1a, $u.id = 000$ has three prefixes: 0, 00 and 000, corresponding to the subtrees rooted at the internal k-d tree nodes with these ids. If we invert the least significant bit of a prefix, we obtain a *maximal sibling subtree*, i.e., a subtree for which there exists no larger subtree that contains it and also not contain the leaf u. Figure 2a shows the maximal sibling subtrees of $u.id = 000$, which are rooted at nodes 1, 01 and 001, as shaded triangles.



(a) Maximal sibling subtrees of u (b) Local knowledge at u

Fig. 2: Links of peer u.

For each maximal sibling subtree, u establishes a link to a peer that resides in it. Note that a subtree may contain multiple leafs and thus multiple peers; MIDAS requires that peer u knows just any one of them. For example, Figure 2a shows the peers in each maximal sibling subtree that u is connected to. Observe that each peer has only partial knowledge about the k-d tree structure. Figure 2b depicts this local knowledge for u, which is only aware about the splits (#1, #2 and #4) along its path to the root. The shaded rectangles corresponds to the subtrees of the same shade in Figure 2a. Peer u knows exactly one other peer within each rectangle. Observe, however, that these rectangles cover the entire space I; this is necessary to ensure that u can locate any other peer, as explained in Section 4.1, and process queries, as discussed in Section 4.

Array $u.link$ defines the routing table. Entry $u.link[j]$ contains the address of a peer that resides in the maximal sibling subtree obtained from the j-length prefix of $u.id$. Continuing the example, u connects to three peers, i.e., $u.link = \{z, w, y\}$. Table 1 depicts the $link$ array for each peer. The notation $u(000)$ indicates that peer u corresponds to k-d tree leaf with id 000. The notation 01: $w(01)$ signifies that peer w

with leaf id 01 is located at the subtree rooted at node 01. The first row of Table 1 indicates that u has three links z, w and y in its maximal sibling subtrees rooted at k-d tree nodes with ids 1, 01 and 001, respectively.

Table 1: Routing tables example

Peer	$link$ **entries**		
u(000)	1: z(101)	01: w(01)	001: y(001)
y(001)	1: z(101)	01: w(01)	000: u(000)
w(01)	1: v(100)	00: u(000)	
v(100)	0: w(01)	11: x(11)	101: z(101)
z(101)	0: y(001)	11: x(11)	100: v(100)
x(11)	0: u(000)	10: v(100)	

3.3 Peer Joins

When a new physical peer joins MIDAS, it becomes responsible for a single peer. Initially, the newly arrived physical peer chooses a uniformly random point p in the space I and locates the peer v responsible for it; Section 4.1 details points query processing. There are two scenarios depending on the status of the physical peer responsible for v.

In the *first scenario*, the physical peer responsible for v has no other peers. Then, the k-d tree leaf node with id $v.id$ is *split* and two new leaves are created. The splitting dimension $sdim$ of the node is chosen uniformly at random among all possible dimensions, while the splitting value $split$ is the value of the random point p on the $sdim$ dimension. Peer v now corresponds to the left child. Finally, a new peer w is created for the right child and is assigned to the newly arrived physical peer.

To ensure proper functionality, MIDAS takes the following actions. (1) v sends to w the tuples that fall in w's zone. (2) Peer v: (2a) appends 0 to $v.id$; (2b) increments $v.depth$ by one; (2c) appends w as the last entry in $v.link$; (2d) appends to $v.sdim$ and $v.split$ the new splitting dimension and value. (3) Peer w: (3a) copies v's state; (3b) changes the least significant bit of $w.id$ to 1; (3c) changes the last entry in $w.link$ to v. (4) v keeps one half of its $v.backlink$. (5) w keeps the other half of its $w.backlink$. (4) w notifies its backlinks about its address.

The *second scenario* applies when the physical peer responsible for v has multiple peers, that is v is just one of them. In this case, v simply migrates to the newly arrived physical peer, which has the responsibility to notify the backlinks of v about its address.

We present an example of how the network of Figure 1 was constructed. Assume initially that there is a single physical peer A responsible for peer u, whose zone is the entire space, as shown in Figure 3a. Then, physical peer B joins and causes a split of the k-d tree root along the first dimension (Split #1 in Figure 1). Peer u is now responsible for the leaf with id 0. A new peer v is created with the id 1 and is assigned to the newly arrived physical peer B. Figure 3b depicts the resulting k-d tree; the split node is drawn with a bold line.

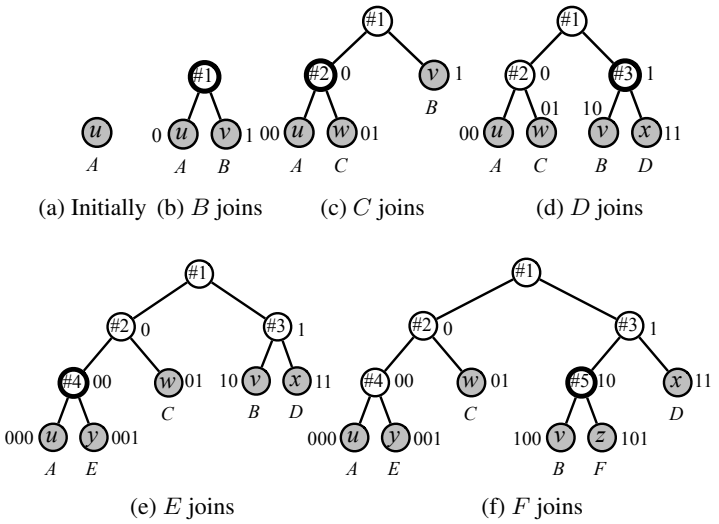


Fig. 3: Network creation.

Assume next that physical peer C joins and chooses a random point that falls in peer u's zone. Therefore, leaf 0 splits, along the second dimension (Split #2). Peer u becomes responsible for the left child and has the id 00, while a new peer w with id 01 is created and assigned to physical peer C. Figure 3c depicts the resulting k-d tree. Then, physical peer D joins selecting a random point inside v's zone. As a result, leaf 1 splits along $v.sdim[2]$ (Split #3), v is assigned leaf 10, and a new peer x with id 11 is assigned to D; see Figure 3d.

Physical peer E arrives and splits leaf 00 along $u.sdim[3]$ (Split #4). Peer u becomes responsible for the left child and obtains the id 000, while a new peer y with id 001 is created and assigned to E. The resulting k-d tree is shown in Figure 3e. Finally, F joins causing a split of leaf 10 along $v.sdim[3]$ (Split #5). A new peer z is assigned to F with id 101, while peer v gets the id 100. Figure 3f shows the k-d tree after the last join.

The following lemma shows that peer joins in MIDAS are *safe*, that is, Lemma 1 continues to hold.

Lemma 2. *After a physical peer joins, the MIDAS invariant holds.*

Proof. Assume that the MIDAS invariant initially holds. In the first scenario, a physical peer join causes a k-d tree leaf to split. Let u be the peer responsible for the leaf that splits, and let u' denote the same peer after the split. Further, let w denote the new peer created. It holds that k-d tree node $u.id$ is the parent of leaves $u'.id$ and $w.id$. Also, note that MIDAS ensures that $I_u = I_{u'} \uplus^{d_u} I_w$. Therefore, any point in the space I that was assigned to u is now assigned to either u' or w, but not to both. All other points remain assigned to the same peer despite the join.

In the second scenario, observe that when a physical peer joins, no changes in the k-d tree and thus in the peers' zones are made. Hence, in both scenarios, the MIDAS invariant is preserved after a physical peer joins. □

The probabilistic nature of the join mechanism in MIDAS achieves a very important goal. It ensures that the (expected value of the) depth of the k-d tree, i.e., the maximum length of a root to leaf path, is logarithmic to the number of total k-d tree nodes (and thus of leaves and thus of peers). The following theorem proves this claim.

Theorem 1. *The expected depth of the distributed k-d tree of MIDAS when n peers join on an initially empty overlay is $O(\log n)$ with constant variance.*

Proof. Consider a MIDAS k-d tree of n peers. Since, each internal node has exactly two children (it corresponds to a split), there are $n - 1$ internal nodes. The k-d tree obtained by removing the leaves is an instance of a *random relaxed k-d tree*, as defined in [8], which is an extension of a *random k-d tree* defined in [2]. This holds because the splitting value and dimension are independently drawn from uniform distributions.

It is shown [8], [2] that the probability of constructing a k-d tree by n random insertions is the same as the probability of attaining the same tree structure by n random insertions into a binary search tree. It is generally known that, in *random binary search trees*, the expected value of a root-to-leaf path length is logarithmic to the number of nodes. However, a stronger result from [16] shows that the *maximum path length*, i.e., the depth, has expected value $O(\log n)$ and variance $O(1)$. This results carries over to the MIDAS k-d tree with n peers. □

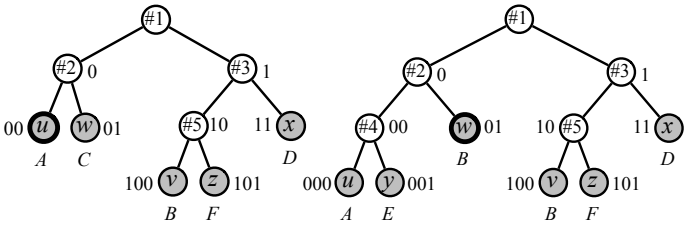
The previous theorem is essential for establishing asymptotic bounds on the performance of MIDAS. First, it implies that the amount of information stored in each peer is logarithmic to the overlay size. Moreover, as discussed in Sections 4.1 and 4, the theorem provides bounds for the cost of query processing.

3.4 Peer Departures

When a physical peer departs, MIDAS executes the following procedure for each of the peers that it is responsible for. Two possible scenarios exist, depending on the location of the departing peer in the k-d tree.

Let y denote a peer of the departing physical peer E in the *first scenario*, which applies when the sibling of y in the k-d tree is also a leaf and thus corresponds to a peer, say u. Observe that y has a link to u, as the last entry in $y.link$ must point to u. In this scenario, when peer y departs, MIDAS adapts the k-d tree by *removing* leaves $y.id$ and $u.id$, so that their parent becomes a leaf. Peer u is properly updated so that it becomes associated with this parent. In the example of Figure 1a, assume that physical peer E, responsible for y, departs. Peer y's sibling is 000, which is a leaf and corresponds to peer u. Figure 4a shows the resulting k-d tree after E departs. Note that peer u is now responsible for a zone which is the union of y's and u's old zones.

To ensure that all necessary changes in this scenario are propagated to the network, MIDAS takes the following actions. (1) y sends to u all its tuples. (2) Peer u: (2a) drops its least significant bit from its id; (2b) decreases its depth by one; and (2c) removes the



(a) E departs (b) C departs

Fig. 4: The two scenarios for peer departures.

last entry from arrays $u.sdim$, $u.split$, $u.link$. (3) y notifies all its backlinks, i.e., the peers that link to y, to update their link to u instead of y. (4) u merges list $y.backlink$ with its own.

Let w be a peer of the departing physical peer C in the *second scenario*, which applies when the sibling of w in the k-d tree is not a leaf. In this case, k-d tree leaf $w.id$ cannot be removed along with its sibling. Therefore, peer w must migrate to another physical peer. Peer w chooses one of its links and asks the corresponding physical peer to assume responsibility for peer w. Ideally, the physical peer that has the lightest load is selected[4] (see also Section 3.6). Note that the backlinks of w must be notified about the address of the new peer responsible for w. In the example of Figure 1a, assume that physical peer C departs. C is responsible for w, whose sibling 00 in the k-d tree is not a leaf. Therefore, w contacts its link v so that its physical peer B assumes responsibility for w. Figure 4b shows the resulting k-d tree after C departs.

The following lemma shows that departures in MIDAS are *safe*, i.e., Lemma 1 continues to hold.

Lemma 3. *After a physical peer departs, the MIDAS invariant holds.*

Proof. Assume that the MIDAS invariant initially holds. Note that a physical peer departure is treated as multiple departures of all peers that it controls.

In the first scenario, a peer departure causes the removal of two k-d tree leaves. Let w be the departing peer and u be the peer responsible for the sibling of $w.id$ in the tree. Further, let u' denote peer u after the departure. Observe that the zone of u' must correspond to some old peer that split along dimension $d_{u'}$, which implies that $I_{u'} = I_u \uplus^{d_{u'}} I_w$. Therefore, any point in the space I that was assigned to either u or w is now assigned to u. All other points remain assigned to the same peer despite the departure.

In the second scenario, observe that when a peer departs, no changes in the k-d tree and thus in the peers' zones are made. Hence, in both scenarios, the MIDAS invariant is preserved after a physical peer departs. □

[4] Peers periodically inform their backlinks about their load.

3.5 Peer Failures

In a dynamic environment, it is common for peers to fail. MIDAS employs mechanisms that ensure that the distributed index continues to function. Consider that a physical peer fails; the following procedure applies for each peer under the responsibility of the failing peer. MIDAS addresses two orthogonal issues when a peer w fails: (1) another physical peer must take over w, and (2) the key-value pairs stored in w must be retrieved.

Regarding the first, note that all peers connected to w will learn that it failed; this happens because a peer periodically pings its neighbors. Each of the peers responsible for one of w's backlinks knows w's zone (i.e., the boundaries of the region for which w is responsible), but only one must take over w. This raises a distributed agreement problem common in other works; for example, in CAN, the backlinks of w would follow a protocol so that the one with the smallest zone takes over w's zone. However, communication among the peers is not necessary in MIDAS. If w's sibling in the tree is a peer (i.e., a leaf), say u, then the physical peer responsible for u will take over w. If that is not the case, the peer with the smallest id, among w's backlinks will take over w.

Regarding the second issue, note that w (or any peer for that matter) is not the owner of the data it stores. Therefore, it is the responsibility of the owner to ensure that its data exist in the distributed index. This is addressed in all distributed indices in a similar manner. Each tuple is associated with a time-to-live (TTL) parameter. The owner periodically (before the TTL expires) re-inserts the tuples in the index. Therefore, the lost key-value pairs of w will eventually be restored. To increase fault tolerance, distributed indices typically employ replication mechanisms. MIDAS is compatible with them as explained in Section 3.6.

3.6 Load Balancing and Fault Tolerance

Balancing the *load*, i.e, the amount of work, among peers is an important issue in distributed indices. MIDAS can use standard techniques. For example, one could apply the task-load balancing mechanism of Chord [19]. That is, given M physical peers, we introduce $N \gg M$ peers. Then each physical peer is assigned a set of peers so that the combined task-load *per physical peer* is uniform.

To enhance *fault tolerance*, MIDAS can utilize standard replication schemes. For example, consider the multiple reality paradigm, where each reality corresponds to an instance of the domain space indexed by a separate distributed k-d tree. Each data tuple has a replica in every reality. A physical peer contains (at least) one peer in each reality. When initiating a query, a physical peer picks randomly a reality to pose the query to; note that this also results in better load distribution. Then, in case of peer failures and before the key-value pairs are refreshed (see Section 3.5), the physical peer can pose the query to another reality.

4 Query Processing on MIDAS

This section details how MIDAS processes multi-attribute queries. In particular, Section 4.1 discusses point queries, while Section 4.2 deals with range queries.

4.1 Point Queries

The distributed k-d tree of MIDAS allows for efficient hierarchical routing. We show that a peer can process a *point query*, i.e., reach the peer responsible for a given point in the space I, in number of hops that is, in expectation, logarithmic to the total number of peers in MIDAS.

Algorithm 1 details how point queries are answered in MIDAS. Assume that peer u receives a point query message for point $\boldsymbol{q}$. If its zone contains $\boldsymbol{q}$, it returns the answer (the value associated with the key $\boldsymbol{q}$) to the issuer, say w, of the query (lines 1–3). Otherwise, u needs to find the most relevant peer to forward the request to. The most relevant peer is the one that resides in the same maximal sibling subtree with the $\boldsymbol{q}$. Therefore, peer u examines its local knowledge of the k-d tree (i.e., the $sdim$ and $split$ arrays) and determines the maximal sibling subtree that $\boldsymbol{q}$ falls in (lines 4–10). The query is then forwarded to the link corresponding to that subtree (line 7).

Algorithm 1 u.Point ($\boldsymbol{q}$, w)
Peer u processes a point query for $\boldsymbol{q}$ issued by w.

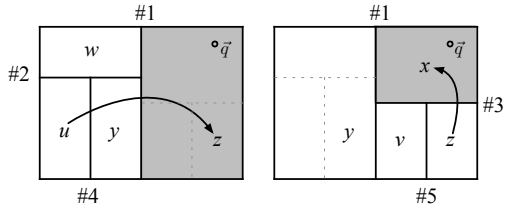
```
1:  if u.IsLocal(q) then
2:      u.Send_to (w, u.Get_val (q))
3:      return
4:  end if
5:  for j ← 0 to u.depth do
6:      d ← u.sdim[j]
7:      if (u.id[j] = 0 and q[d] ≥ u.split[j]) or (u.id[j] = 1 and q[d] < u.split[j]) then
8:          u.link[j].Point (q, w)
9:          return
10:     end if
11: end for
```

To illustrate the previous procedure, consider a query for point $\boldsymbol{q}$ issued by peer u. Figure 5a draws $\boldsymbol{q}$ and the local knowledge about the space I at peer u. Observe that $\boldsymbol{q}$ falls outside u's zone. Peer u thus forwards the query to its link z within the shaded area since it contains $\boldsymbol{q}$ (1st hop). Next, peer z processes the query. Point $\boldsymbol{q}$ is inside the shaded area of Figure 5a, which depicts z's local k-d tree knowledge. Subsequently, z forwards the query to its link x within that area (2nd hop). Finally, peer x responds to the issuing peer u, as point $\boldsymbol{q}$ falls inside its zone.

Lemma 4. *The expected number of hops in a point query is* $O(\log n)$.

Proof. We first show that the number of hops required is at the worst case equal to the depth of the k-d tree.

Assume that the requested point is $\boldsymbol{q}$. Consider a peer u that executes Algorithm 1. u determines the maximal sibling subtree that $\boldsymbol{q}$ resides in, and let k be the depth of its root. Then, u forwards the request to peer v in that subtree. We argue that v will determine a maximal sibling subtree at depth $\ell > k$. Observe that u and $\boldsymbol{q}$ fall in the same subtree rooted at depth k. Therefore, all subtrees rooted at depths higher than k that contain u will also contain $\boldsymbol{q}$. The argument holds because all maximal sibling



(a) Processing at u (b) Processing at z

Fig. 5: Point query example for $\boldsymbol{q}$.

subtrees of u rooted at depths higher than k cannot contain $\boldsymbol{q}$. The above argument implies that point queries will be forwarded to subtrees of successively higher depth, until the leaf which has $\boldsymbol{q}$ is reached; such a leaf exists because $\boldsymbol{q} \in I$. Therefore, the number of hops is at the worst case equal to the depth of the k-d tree.

From Theorem 1, we have that the expected depth of the k-d tree is $O(\log n)$, which concludes the proof. □

4.2 Range Queries

A range query specifies a rectangular area Q in the space, defined by a lower $\boldsymbol{\ell}$ and a higher point $\boldsymbol{h}$, and requests all tuples that fall in Q. Instead of locating the peer responsible for a corner of the area, e.g., $\boldsymbol{\ell}$, and then visit all relevant neighboring peers, MIDAS utilizes the distributed k-d tree to identify in parallel multiple peers whose zone overlaps with Q. The range partitioning idea is similar to the shower algorithm [7], which however applies only for single-dimensional data.

Algorithm 2 details the actions taken by a peer u upon receipt of a range query for area $Q = [\boldsymbol{\ell}, \boldsymbol{h}]$ issued by w. First, u identifies all its tuples inside Q, if any, and sends them to the issuer w (lines 1–3). Then, u examines all its maximal sibling subtrees by scanning arrays $sdim$ and $split$ (lines 4–15). If the area of a subtree overlaps Q (lines 6 and 10), peer u constructs the intersection of this area and Q (lines 7–8 and 11–12). Then, u forwards a request for this intersection to its link (lines 9 and 13). Lines 6–9 (resp. 10–14) apply when u is in the left (resp. right) subtree rooted at depth i.

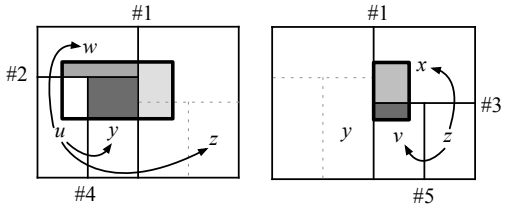
Figure 6 illustrates an example of a range query issued by peer u. Initially, u executes Algorithm 2 for the range depicted as a bold line rectangle in Figure 6a. Peer u retrieves the tuples inside its zone that are within the range; these tuples reside in the non-shaded region of the range in Figure 6a. Then, u constructs the shaded regions, shown in Figure 6a, as the intersections of the range with the area corresponding to its maximal sibling subtrees. For each of these shaded regions, u forms a new query and sends it to the appropriate link (1st hop); the messages are depicted as arrows in Figure 6. For instance, peer z, which is u' link in the maximal sibling subtree rooted at depth one, receives a query about the light shaded area.

Peers w, y and z receive a query from u. The range for w and y falls completely within their zone. Therefore, they process them locally and do not send any other message. Figure 6b illustrates query processing at peer z, where the requested range is drawn as a bold line rectangle. Observe that this range does not overlap with z's zone; therefore, z has no tuple that satisfies the query. Then, z constructs the intersections of

Algorithm 2 u.Range $(\boldsymbol{\ell}, \boldsymbol{h}, w)$
Peer u processes a range query for rectangle $Q = [\boldsymbol{\ell}, \boldsymbol{h}]$ issued by w.

```
1: if u.Overlaps (ℓ, h) then
2:     u.Send_to (w, u.Get_vals (ℓ, h))
3: end if
4: for j ← 0 to u.depth do
5:     d ← u.sdim[j]
6:     if u.id[j] = 0 and u.split[j] < h[d] then
7:         ℓ' ← ℓ
8:         ℓ'[d] ← u.split[j]
9:         u.link[j].Range (ℓ', h, w)
10:    else if u.id[j] = 1 and u.split[j] > ℓ[d] then
11:        h' ← h
12:        h'[d] ← u.split[j]
13:        u.link[j].Range (ℓ, h', w)
14:    end if
15: end for
```

the range with the areas in its maximal sibling subtrees. Observe that the range does not overlap with the maximal sibling subtree rooted at depth one; hence, no peer receives a duplicate request. Peer z sends a query message to its links v and x with the shaded regions of Figure 6b (2nd hop). Finally, peers v and x process the queries locally as the requested ranges have no overlap with their zones.



(a) Processing at u (b) Processing at z

Fig. 6: Range query example.

As explained in the example of Figure 6, the query is answered in two hops. In the first, u reaches y, w, and z, and in the second, z reaches v and x. The following lemma shows that the expected number of hops is logarithmic to the number of peers.

Lemma 5. *The expected number of hops for processing a range query is* $O(\log n)$.

Proof. We show that the number of hops required is at the worst case equal to the depth of the k-d tree.

Consider that virtual peer v receives from u a query for range $Q = [\boldsymbol{\ell}, \boldsymbol{h}]$; assume that v is the link at depth k in the $link$ array of u. Due to its construction (the result of an intersection operation), range Q is completely contained within the subtree that contains v rooted at depth k. Therefore, Q cannot intersect with any maximal sibling

subtree of v at depth lower than k. As a result, if v forwards a range query, it will be to links at depths strictly higher than k.

In the worst case, a message will be forwarded to as many virtual peers as the depth of the k-d tree. The fact that the expected depth is $O(\log n)$ (from Theorem 1) completes the proof. □

5 Experimental Evaluation

In order to assess our methods and validate our analytical results, we simulate a dynamic environment and study each query type. We implement two methods from the literature, CAN [15] and the VBI-tree [11], to serve as competitors to MIDAS.

5.1 Setting

Network We simulate a dynamic topology to capture arbitrary peer joins, and departures, by implementing two distinct stages. In the *increasing stage*, peers continuously join the network, while no peer departs. It starts from an overlay of 1,000 peers and ends when 100K peers are available. On the other hand, in the *decreasing stage*, peers continuously leave the network, while no new peer joins. This stage starts from an overlay of 100K peers and ends when only 1,000 peers are left. When depicting the effect of these stages in the figures, the solid (resp. dashed) line represents the increasing (resp. decreasing) stage.

Data and Queries We use both a *real* as well as *synthetic* datasets of varying dimensionality. The real dataset, denoted as *NE*, consists of spatial (2D) points representing 125K postal addresses in three metropolitan areas (New York, Philadelphia and Boston). The synthetic datasets contain 1M tuples uniformly and independently distributed in the domain space. The dimensionality is varied from 2 up to 19.

For each point query, we choose uniformly and independently a random location in the domain space. For each range query, we also choose a random location, while the length of the rectangular sides are selected so that the query returns approximately 50 tuples. In all figures, the reported values are the averages of executing 50K queries over 10 distinct overlay topologies.

Performance Metrics We employ several metrics to evaluate MIDAS against other methods. The basic metric indicative of query performance is *latency*, which is defined as the maximum distance (in terms of hops) from the issuing peer to any peer reached during query processing. Clearly, lower values suggest faster response.

Distributed query processing imposes a task load on multiple peers. Two metrics quantify this load. *Precision* is defined as the ratio of the number of peers that contribute to the answer over the total number of peers reached during processing of a query; the optimal precision value is 1. *Congestion* is defined as the average number of queries processed at any node, when n uniformly random queries are issued (n is the number of peers in the network); lower values suggest lower average task load.

The next metrics are independent of the query evaluation process. Note that two types of information are stored locally in each peer. The first is overhead information (e.g., links, zone description, etc.), which is measured by the average *state* a peer must maintain. As the number of peers increase, this is an important measure of scalability. The second type is the number of tuples stored. As this depends on the dataset distribution and the network topology, imbalances can occur. *Data load* measures the percentage of the total data in the network, that resides in the top $Q\%$ most loaded peers. We show measurements for $Q = 10\%$, where the optimal fairness value is 0.1.

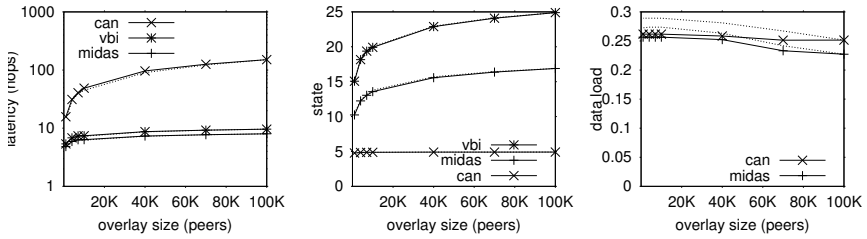
5.2 Results

This section presents the findings of our experimental study. Figure 7a illustrates query performance aspects for point queries for spatial workloads, where MIDAS clearly outperforms the competition. Latency for MIDAS is bounded by $O(\log n)$, where n stands for the overlay size, as Lemma 4 predicts; whereas, latency for CAN is bounded by $O(\sqrt[d]{n})$. Figure 7b depicts the average state of a node maintains, as it is directly related to the amount of traffic that occurs due to maintenance operations like detecting failures, preserving updated routing tables each time a peer joins or leaves. Note that state is increased in the VBI-tree compared to MIDAS as peers keep information about the peers on their path from the root and their siblings at the same depth of the tree. State in MIDAS has a logarithmic behavior in terms of the overlay size, as Theorem 1 states. In low dimensionality settings, such as the spatial dataset *NE*, CAN peers maintain very few links to others. Moreover, Figure 7c presents the data load. CAN achieves lower data load mainly due to its joining protocol. In particular, it chooses to halve a peer's area for a newcomer, instead of splitting its data load like in MIDAS. As a result, CAN becomes vulnerable to data skew. Note that in all methods, data load slightly decreases as the number of peers increases.

We next discuss the case of range queries. Latency shows similar logarithmic behavior with range queries (Figure 8a) as Lemma 5 predicts. Figure 8b illustrates that the congestion in MIDAS has logarithmic behavior in terms of the overlay size, and is significantly lower than its competitors.

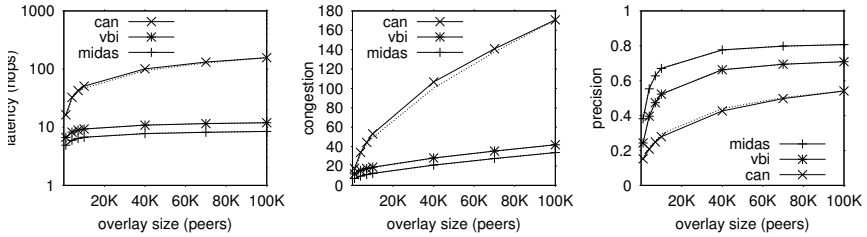
In Figure 8c precision improves with overlay size. The reason is that peers become responsible for smaller areas, while the queries have fixed size, in other words less irrelevant peers are reached. Precision is worse for VBI-tree and CAN compared to MIDAS because of the longer routes required to reach relevant peers (see latency).

We finally discuss synthetic datasets of varying dimensionality. As Figures 9a and 9b, MIDAS is largely unaffected by dimensionality and is asymptotically better than both the VBI-tree and CAN. The expected latency for CAN is bounded by $O(\sqrt[d]{n})$, and thus, decreases as dimensionality increases. However, this comes at an extra cost. As Figure 9c shows, the expected cardinality of a node's routing table increases with dimensionality and it is confirmed to be in $O(d)$, whereas MIDAS' peer state is in $O(\log n)$ regardless dimensionality degree.



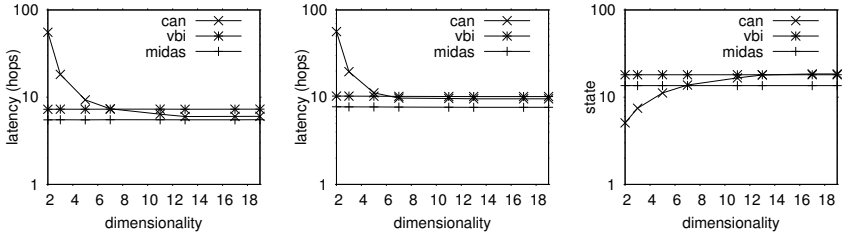
(a) latency (point queries) (b) state (c) data load

Fig. 7: Latency, state cardinality a peer maintains, and data load for the NE dataset.



(a) latency (range queries) (b) congestion (range queries) (c) precision (range queries)

Fig. 8: Latency, congestion, and precision for the NE dataset.



(a) latency (point queries) (b) latency (range queries) (c) state

Fig. 9: Latency, cardinality of the state a peer maintains for multidimensional dataset.

6 Conclusion

In this work we have presented MIDAS, a pure multidimensional indexing scheme for large-scale decentralized networks, which significantly differs from other popular overlays which are either single dimensional or implement a space filling curve. Our peer-to-peer variant offers the possibility of combining DHT with hierarchical space partitioning schemes, avoiding order-preserving hashing and space-filling curves. Yet, it outperforms other multidimensional structures in terms of scalability. MIDAS allows for multidimensional queries and offers guarantees concerning all operations. In particular, updates, points and range queries are resolved in $O(\log n)$ hops. Most importantly, each peer in MIDAS maintains $O(\log n)$ state only. All things considered, MIDAS constitutes an extremely attractive solution when it comes to high dimensional datasets that

provides a rich and wide functionality. Finally, interesting results arose from this work. The curse of dimensionality has no impact on query performance and maintenance costs, while MIDAS achieves high levels of fairness.

References

1. K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-grid: a self-organizing structured p2p system. *SIGMOD Record*, 32(3):29–33, 2003.
2. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.
3. J. L. Bentley. K-d trees for semidynamic point sets. In *Symposium on Computational Geometry*, pages 187–197, 1990.
4. A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM*, pages 353–366, 2004.
5. S. Blanas and V. Samoladas. Contention-based performance evaluation of multidimensional range search in p2p networks. In *InfoScale'07*, pages 1–8, 2007.
6. M. Cai, M. R. Frank, J. Chen, and P. A. Szekely. Maan: A multi-attribute addressable network for grid information services. *J. Grid Comp.*, 2(1):3–14, 2004.
7. A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range queries in trie-structured overlays. In *P2P Computing*, pages 57–66, 2005.
8. A. Duch, V. Estivill-Castro, and C. Martínez. Randomized k-dimensional binary search trees. In *ISAAC*, pages 199–208, 1998.
9. P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multidimensional queries in p2p systems. In *WebDB*, pages 19–24, 2004.
10. H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *VLDB*, pages 661–672, 2005.
11. H. V. Jagadish, B. C. Ooi, Q. H. Vu, R. Zhang, and A. Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *ICDE*, page 34, 2006.
12. D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symp. on Theory of Comp.*, pages 654–663, 1997.
13. P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS*, pages 53–65, 2002.
14. C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory Comput. Syst.*, 32(3):241–280, 1999.
15. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01*, pages 161–172, 2001.
16. B. A. Reed. The height of a random binary search tree. *Journal of the ACM*, 50(3):306–332, 2003.
17. A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.
18. Y. Shu, B. C. Ooi, K.-L. Tan, and A. Zhou. Supporting multi-dimensional range queries in peer-to-peer systems. In *Peer-to-Peer Computing*, pages 173–180, 2005.
19. I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable p2p lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.
20. B. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Comm.*, 22(1):41–53, 2004.