MIDAS: Multi-Attribute Indexing for Distributed Architecture Systems

George Tsatsanifos [#1], Dimitris Sacharidis [*2], Timos Sellis [#*3]

[#] *School of Electric and Computer Engineering*
National Technical University of Athens
[1] gtsat@dblab.ntua.gr

[*] *Institute for the Management of Information Systems, R.C. "Athena"*
[2] dsachar@imis.athena-innovation.gr
[3] timos@imis.athena-innovation.gr

***Abstract*— This work presents a novel pure multi-dimensional, 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 multi-dimensional lookups, range search, proximity and K nearest neighbors search. Scalability is enhanced as each peer has only partial knowledge of the network. The most prominent feature of our method, is that 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 a simple, efficient mechanism in order to compensate for unexpected peer failures. Finally, our work is complemented by an experimental evaluation, where MIDAS is compared to other known methods.**

I. 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 (lookups), range, and nearest neighbor queries, in arbitrarily high 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 separately, e.g., [1], [2]. However, these approaches still have to resort to only one of the dimensions for processing queries. The most popular approach, [3], [4], [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., [6], [3]. 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 all intermediate node, i.e., jumps cannot be made.

The last category includes methods, e.g., [7], [8], 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 [8] showed that for dimensionality close to 20, this method was outperformed by the non-indexed approach of [6].

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 [9] for single dimensional tree-like structures. In addition, k-d trees are known to efficiently process nearest neighbor queries in arbitrary high dimensionality [10].

Unlike other multidimensional index techniques, e.g., [8], 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, this allows MIDAS to have a simple load balancing mechanism: a new peer that joins is routed to the leaf with the heaviest load and causes it to split, halving thus its area of responsibility.

In summary, MIDAS is an efficient method for indexing multi-attribute data. We prove that lookups (i.e., point queries) and range queries are performed in $O(\log n)$ hops, while nearest neighbor search in $O(\log^2 n)$ hops. These bounds are smaller than non-indexed multidimensional P2P systems, e.g., $O(d\sqrt[d]{n})$ of [6], and a thorough experimental study validates this claim, demonstrating that MIDAS consistently outperforms [6] even in high dimensionality.

The remainder of this paper is organized as follows. Section II compares MIDAS to related work. Section III describes our index scheme and basic operations including fault-tolerance issues. Section IV discusses multidimensional query processing. Section V presents an extensive experimental evaluation of all MIDAS' features. Section VI concludes and summarizes our contributions.

II. 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 [11] 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 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, anycast, domain name services, etc.

Chord [12] 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 [13], Kademlia [14], Tapestry [15] and Pastry [16]. Peer lookups in these systems is based on Plaxton's algorithm [9]. 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 multi-dimensional lookups in $O(\log n)$ hops.

We next discuss various structured peer-to-peer systems that index multi-attribute keys and are thus able to process range and K-NN queries. In CAN [6], 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 [3], 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 [3], ZNet [4], employ a space filling curve to map the multi-dimensional 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 [2] extends Chord to support multi-dimensional range queries by mapping attribute values to the Chord identifier space via uniform locality preserving hashing. MAAN and Mercury [1] can support multi-attribute range queries through single-attribute query resolution. They do not feature pure multi-dimensional 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. Additionally, NN search is not discussed in these works at all.

The VBI-tree [8] is a peer-to-peer framework based on balanced multi-dimensional tree structured overlays, e.g., the R-tree. It provides an abstract tree structure on top of an overlay network, which can support 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.

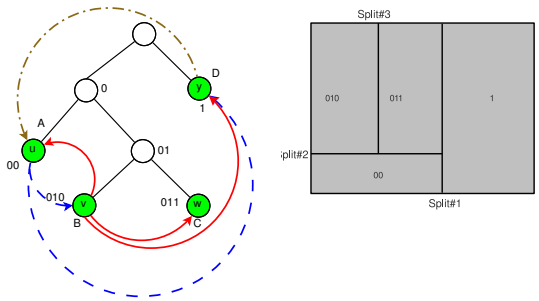
III. MIDAS Architecture

This section presents the information stored in each peer and details the basic operations in the MIDAS overlay network. In particular, Section III-A introduces the distributed index structure, Section III-B discusses the peers in MIDAS, Section III-C, III-D and III-E elaborates on the actions taken when a peer departs, joins, and fails, respectively, and Section III-F details the lookup procedure.

A. Index Structure

The distributed index of MIDAS is an instance of an adaptive k-d tree [17]. Consider a D-dimensional space $I = [\vec{\ell}_I, \vec{h}_I]$, defined by a low $\vec{\ell}_I$ and a high $\vec{h}_I$ 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 criteria (i.e., the values of s_i and d_i) will be discussed shortly. Note that d_i represents the splitting dimension of node $T[i]$ and not the i-th dimension of the space. Therefore, for $I_{2i} = [\vec{\ell}_{2i}, \vec{h}_{2i}]$ and $I_{2i+1} = [\vec{\ell}_{2i+1}, \vec{h}_{2i+1}]$, it holds that (1) $\vec{\ell}_{2i}[d_j] = \vec{\ell}_{2i+1}[d_j]$ and $\vec{h}_{2i}[d_j] = \vec{h}_{2i+1}[d_j]$ on every dimension $d_j \neq d_i$, and (2) $\vec{h}_{2i}[d_i] = \vec{\ell}_{2i+1}[d_i] = s_i$ on dimension d_i, assuming that the left child $T[2i]$ is assigned the lower part of I_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 can be assigned a binary identifier corresponding to its path from the root, 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 seven nodes with three splits and their ids. 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 four leaves drawn green in Figure 1a.



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

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

A tuple with D attributes can be seen as a point in D-dimensional space, and thus k-d trees can be used to index a set of tuples. Tuples are only stored at the leaves. A leaf stores all tuples that fall in its rectangle. The hierarchical structure of the k-d tree allows for efficient methods to locate all tuples within a range, and to retrieve the K nearest neighbors of a given tuple.

A final note concerns the splitting criteria of a k-d tree. There exist numerous variations, but in this work we assume the following. The splitting dimension of a node is the one with the highest spread in its domain. The splitting value is selected so that each child has roughly the same number of tuples that fall in its rectangle. Typically, nodes are split until there is a single tuple in each leaf. However, as we discuss in Section III-B, a leaf node corresponds to a virtual peer in MIDAS, and thus it is allowed to store multiple tuples.

B. MIDAS Peers

A *virtual 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*, i.e., an actual machine in the distributed network, may act as several virtual peers due to node departures or fails as we discuss in Section III-C. A virtual peer is denoted with small letters, e.g., u, v, w, etc., whereas a peer with capital letters, e.g., A, B, C, etc. For example, in Figure 1a, peer W acts as a single virtual peer w corresponding to leaf 011. We emphasize that internal k-d tree nodes, e.g., 0, 01, do not correspond to peers (virtual or actual).

A virtual peer v contains the following information about the k-d tree and other peers. (1) $v.id$ is a bitmap representing the leaf's binary id; $v.id[j]$ is the j-th most significant bit. (2) $v.depth$ is the depth of the leaf in the k-d tree, or equivalently the number of bits in $v.id$. (3) $v.sdim$ is an array of length $v.depth$ so that $v.sdim[j]$ is the splitting dimension of the parent of the j-th node on the path from the root to v. (4) $v.split$ is an array of length $v.depth$ so that $v.split[j]$ is the splitting value of the parent of the j-th node on the path from the root to v. (5) $v.link$ is an array of length $v.depth$ that corresponds to v's routing table, i.e., it contains the virtual peers v has a link to. (6) $v.backlink$ is a list that contains all virtual peers that have v in their $link$ array.

In the following, we discuss the routing table of a virtual peer v. Consider the prefixes of v's identifier; there are $v.depth$ of them. Each prefix corresponds to a subtree of the k-d tree that contains the leaf v (more accurately the leaf that has id $v.id$) and identifies a node on the path from the root to v. In the example of Figure 1a, $v.id = 010$ has three prefixes: 0, 01 and 010, 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 v. For $v.id = 010$ the maximal sibling subtrees are rooted at nodes 1, 00 and 011.

For each maximal sibling subtree, v establishes a link to a virtual peer that resides in it. Note that a subtree may contain

multiple leafs and thus multiple virtual peers; MIDAS requires that v knows any one of them. Array $v.link$ stores the routing table. Entry $v.link[j]$ links to a virtual peer that resides in the subtree corresponding to a j-length inverted prefix of $v.id$.

Continuing the example, v connects to three virtual peers, i.e., $v.link = \{y, u, w\}$; the links are drawn solid red arrows in Figure 1a. Table I depicts the $link$ array for each peer. The notation $u(00)$ indicates that virtual peer u has the id 00. The notation 1: $y(1)$ signifies that y with id 1 is located at the subtree rooted at node 1. Therefore, the first row indicates that u has two links y and v.

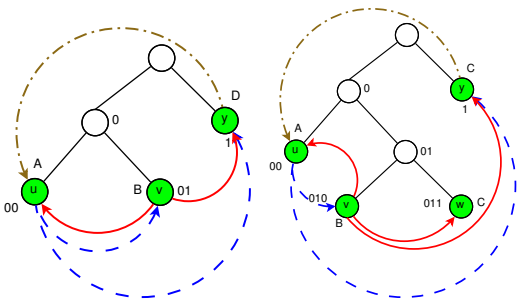
TABLE I: Routing tables example

Peer	*link* **entries**		
$u(00)$	1: $y(1)$	01: $v(010)$	
$v(010)$	1: $y(1)$	00: $u(00)$	011: $w(011)$
$w(011)$	1: $y(1)$	00: $u(00)$	010: $v(010)$
$y(1)$	0: $u(00)$		

C. Peer Departures

We discuss the actions taken when a peer departs. Since the process for each virtual peer of the departing peer is identical, we only need to discuss the case for a single virtual peer. There are two possible scenarios.

Assuming w is a virtual peer of the departing peer, the first scenario applies when the sibling of w in the k-d tree is also a leaf. In this case, w has a link to the virtual peer, say v, corresponding to its sibling. In other words, the last entry of $w.link$ points to v. When w departs, MIDAS conceptually adapts the k-d tree by *merging* leaves w and v. Their parent, which is now a leaf, becomes the k-d tree node associated with the virtual peer v. In the example of Figure 1a, w's sibling is 010, which is a leaf. Figure 2a shows the resulting k-d tree after virtual peer w departs.



(a) w departs (b) y departs

Fig. 2: Two scenarios of virtual peer departures.

To ensure that all necessary changes due to w's merge into v are propagated to the network, MIDAS takes the following actions. (1) w sends to v all its tuples. (2) virtual peer v: (2a) drops its least significant bit from its id; (2b) decreases its depth by one; and (2c) removes the last entry from arrays $v.sdim$, $v.split$, $v.link$. (3) w notifies all its backlinks, i.e., the virtual peers that link to w, to update their link to v instead. (4) v merges list $w.backlink$ with its own.

Assuming y is a virtual peer of the departing peer, the second scenario applies when the sibling of y in the k-d tree is not a leaf. In this case, there is no virtual peer to merge with y in a manner that results in a valid k-d tree. Therefore, w must be handled by some other (actual) peer. So, y selects one of its links and notifies that peer to become responsible for virtual peer y. Ideally, the link that has the lightest load is selected*. Note that the backlinks of y must be notified about the address of the new peer responsible for y. In the example of Figure 1a, y's sibling is 0, which is not a leaf. Figure 2a shows the resulting k-d tree after virtual peer y departs.

The following lemma shows that there always exists a virtual peer responsible for every part of the space I, and thus for each tuple, even after multiple departures.

Lemma 1: Consider a MIDAS network of n peers indexing space I. Then, after $k < n$ virtual peers depart, there is no subspace of I for which no peer is responsible.

Proof: Departures of the second scenario do not affect the virtual peers; a virtual peer simply migrates from the departing peer to another. Therefore, we only need to prove the lemma for k departures of the first scenario. We prove by induction.

- For $k = 1$, let the departing peer u which shares its longest common prefix with peer w. Then, u will merge with w into w', retracting this way the virtual k-d tree. By construction, when merging peers u and w, then $I_{w'} = I_u \uplus^{d_u} I_w$. This practically means that after u's departure peer the merged peer w' will be responsible for $I_{w'} = I_w \uplus^{d_u} I_u$ and all the associated data keys. Hence, $I = \underbrace{\biguplus_{v=1}^{n} I_v = (I_w \uplus^{d_w} I_u) \uplus^{d_u} \biguplus_{v=1}^{n-2} I_v}_{\text{before merging}} = \underbrace{I_{w'} \uplus^{d_{w'}} \biguplus_{v=1}^{n-2} I_v = \biguplus_{v=1}^{n-1} I_v}_{\text{after merging}}$†
- For $k = m$, we assume that it is true. Hence, we take $\underbrace{(((I_m \uplus^{d_m} \cdots) \uplus^{d_2} I_1) \uplus^{d_1} I_{w'})}_{\text{applicable merging sequence}} \uplus^{d_{w'}} \biguplus_{v=1}^{n-m-1} I_v = \biguplus_{v=1}^{n-m} I_v = I$, which corresponds to the initial coordinate Euclidean space partitioned by n peers.
- For $k = m+1$ and the same scenario of departures $I = \underbrace{((((I_{m+1} \uplus^{d_{m+1}} I_m) \uplus^{d_m} \cdots) \uplus^{d_2} I_1) \uplus^{d_1} I_{w'})}_{\text{an applicable merging sequence}} \uplus^{d_{w'}} \biguplus_{v=1}^{n-m-1} I_v = \biguplus_{v=1}^{n-m} I_v$

Thus, after any number of departures, there is no subspace of I for which no virtual peer is responsible. ■

D. Peer Joins

When a new peer, say E, joins MIDAS, it will become responsible for a single virtual peer. Initially, the new peer

*Peers periodically inform their backlinks about their load

†Where the v iterator takes such values that the $\bigcup_{v=1}^{n-k}$ corresponds to applicable departures, and therefore compatible union of responsibility areas.

chooses a random point (tuple) in the space I and locates the virtual peer, say v, responsible for it; Section III-F explains the lookup procedure. There are two scenarios.

In the first scenario, the peer responsible for v has no other virtual peers. In this case, the k-d tree leaf v is split and two new leaves are created; the splitting criteria are those described in Section III-A. v now corresponds to the left child, whereas a new virtual peer w is created for the right child. The new virtual peer w is finally assigned to the arriving peer E.

To ensure proper functionality, MIDAS takes the following actions. (1) v sends to w the tuples that fall in w's rectangle. (2) Virtual 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) Virtual 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 E's address.

The second scenario applies when the peer responsible for v has multiple virtual peers; i.e., v is just one of them. In this case, v simply migrates to the new peer E. Then, E needs to notify the backlinks of v about its address.

The following lemma shows that tuple is lost after a series of peer joins.

Lemma 2: Consider a MIDAS network of a single peer indexing the entire space I. Then, after k virtual peers join, there is no subspace of I for which no peer is responsible.

Proof: Joins of the second scenario do not affect the virtual peers, as they simply migrate among peers. We prove the lemma for k joins of the first scenario. We prove by induction.

- For $k = 1$, let the newly inserted virtual peer w, which has acquired half of u's load along the most spread dimension d_u. By construction $I_u \uplus^{d_u} I_w = I$, which corresponds to the initial coordinate Euclidean space.
- For $k = n$, we assume it is true. Thus, $\biguplus_{v=0}^{n} I_v = I$.[‡]
- For $k = n + 1$, let w be the $n + 1$-th virtual peer that joined splitting u's zone on dimension d_u. For ease of presentation, we assume that u was the n-th virtual peer to join. Hence, $\biguplus_{v=0}^{n+1} I_v = (I_w \uplus^{d_u} I_u) \uplus^{d_{n-1}} \biguplus_{v=0}^{n-1} I_v$. The union of the zones of peers u and w corresponds to I_u, and from the $k = n$ step $\biguplus_{v=0}^{n} I_v = I$. Hence, we take that $\biguplus_{v=0}^{n+1} I_v = I$, which corresponds to the initial coordinate Euclidean space.

Thus, after any number of virtual peer joins there is no subspace of I for which no peer is responsible. ■

We present an example of how the network of Figure 1 was constructed. Assume initially that only one virtual peer u, which is responsible for the whole space, exists. Then, y joins and thus u is split along the first dimension (Split#1 in Figure 1). The responsibility for the higher part of the area and the associated tuples are assigned to y. With this operation, u and y will obtain their new ids 0 and 1, respectively. When another virtual peer v joins MIDAS, it causes another split on u along the second dimension (Split#2), and will take over the higher part. As a result, peer u corresponds to the k-d tree leaf 00, and v to 01. Finally, when w joins, it splits v's along the first dimension (Split#3), and virtual peers v, w become responsible for leaves 010 and 011.

Peer joins are the basic mechanism for ensuring load balancing in MIDAS. The first step in joining is choosing a random point of the space and thus selecting a virtual peer. A virtual peer with large zone is more likely to be selected by this procedure. Similarly, a peer responsible for multiple virtual peers is also selected with a large probability. Therefore, after a large number of joins, each peer will have a single virtual peer with roughly the same zone size (i.e., load), and the k-d tree will be a complete binary tree. As a result the k-d tree will have $O(n)$ internal nodes and $O(n)$ leaves (and thus peers). The expected maximum depth of a peer is $O(\log n)$, which determines the amount of information stored in each peer, and the number of hops necessary to lookup a peer and process range and NN queries.

E. Peer Failures

In an extremely dynamic environment, it is very common for peers to fail. Therefore, MIDAS employs mechanisms that enable proper reconstruction of the overlay topology and ensure consistency so that no peer become unreachable and no zone disappears.

Assume that the peer responsible for virtual peer w fails, and therefore u is unable to reach w from u's j-link. Then, u responds according to the two following scenarios. If peer w is the last link of u's routing table, then u takes over w's zone, by expanding its own appropriately and informing its new sibling about the change. Therefore, we need to assume that peer u knows in advance w's zone. Otherwise, u bypasses the unreachable virtual peer w by issuing a lookup query for a random point, corresponding to the area designated by the j-th split point. Besides, if no answer is returned to u (e.g., the query point falls into the area of the failed peer), then it can repeat the procedure for a different point. By processing the answer, u will retrieve virtual peer v and will replace its j-link w with v.

Note that when a peer fails, its tuples get lost. This a common issue in structure peer-to-peer systems and a simple solution is to periodically refresh the tuples stored at the network by the data owner.

F. Peer lookups

The distributed k-d tree of MIDAS allows for efficient hierarchical routing. We show that a virtual peer can be reached from any other in hops, i.e., the *routing diameter* of MIDAS is, logarithmic to the number of peers.

Algorithm 1 details how lookup queries are answered in MIDAS. Virtual peer u receives a lookup query message for point $\vec{q}$. If its zone contains $\vec{q}$, it returns the answer to the issuer w of the query (lines 1–3). Otherwise, u needs to find the most

[‡]The iterator v in the $\bigcup_{v=0}^{k}$ union takes values corresponding to the reverse order that the peers joined the overlay, and therefore compatible zone unions.

relevant peer to forward the request to. The most relevant peer is the one that resides in the same maximal sibling subtree with the $\vec{q}$. Therefore, virtual 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 $\vec{q}$ falls in (lines 4–10). The query is then forwarded to the link corresponding to that subtree (line 7).

Algorithm 1 u.Lookup ($\vec{q}, w$): Peer u processes a lookup query for $\vec{q}$ issued by w

Require: $\vec{q} \in I$
1: **if** u.IsLocal($\vec{q}$) **then**
2: ReturnAnswer (u.getVal($\vec{q}$), w)
3: **end if**
4: **for** $j = 0$ to $u.depth - 1$ **do**
5: $d = u.sdim[j]$ /* splitting dimension */
6: **if** ($u.id[j] = 0$ and $\vec{q}[d] \geq u.split[j]$) or ($u.id[j] = 1$ and $\vec{q}[d] < u.split[j]$) **then**
7: $u.link[j]$.Lookup ($\vec{q}$, w)
8: return j
9: **end if**
10: **end for**

Lemma 3: The expected number of hops in a lookup query is $O(\log n)$ in the worst case.

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

Assume that the requested point is $\vec{q}$. Consider a virtual peer u that executes Algorithm 1. u determines the maximal sibling subtree that $\vec{q}$ resides in, and let k be the depth of its root. Then, u forwards the request to virtual peer v in that subtree. We argue that v will determine a maximal sibling subtree at depth $\ell > k$. Observe that u and $\vec{q}$ fall in the same subtree rooted at depth k. Therefore, all subtrees rooted at depths higher than k that contain u will also contain $\vec{q}$. The argument holds because all maximal sibling subtrees of u rooted at depths higher than k cannot contain $\vec{q}$. The above argument implies that lookup queries will be forwarded to subtrees of successively higher depth, until the leaf which has $\vec{q}$ is reached; such a leaf exists because $\vec{q} \in I$. Therefore, the number of hops is at the worst case equal to the maximum depth of the k-d tree.

As discussed in Section III-D, after a sufficient number of joins, the k-d tree will be a complete binary tree. Thus, its expected maximum depth is $O(\log n)$. ■

IV. Query Processing on MIDAS

This section details how MIDAS processes multi-attribute queries. In particular, Section IV-A discusses range queries, while Section IV-B deals with nearest neighbor queries.

A. Range Queries

A range query specifies a rectangular area Q in the space, defined by a lower $\vec{\ell}$ and a higher point $\vec{h}$, and requests all tuples that fall in Q. Instead of locating the peer responsible for a corner of the area, e.g., $\vec{\ell}$, and then visit all relevant neighboring peers, MIDAS utilizes the distributed k-d tree to identify in parallel multiple virtual peers whose zone overlaps with Q. The range partitioning idea is similar to the shower algorithm [18] for single-dimensional data; however, due to its index, MIDAS is capable of processing multi-attribute range queries efficiently.

Algorithm 2 details the actions taken by a peer u upon receipt of a range query for area $Q = [\vec{\ell}, \vec{h}]$ 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), virtual 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). Note that lines 6–9 (resp. 10–14) apply when u is in the left (resp. right) subtree rooted at depth j.

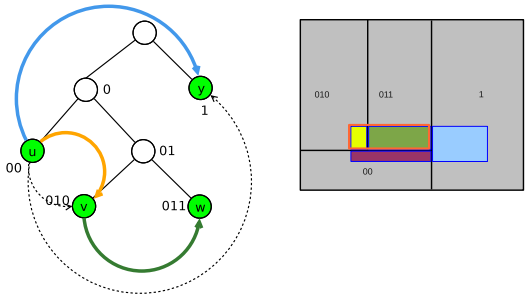
Algorithm 2 u.Range ($\vec{\ell}, \vec{h}, w$): Peer u processes a range query for rectangle $Q = [\vec{\ell}, \vec{h}]$ issued by w

Require: $\vec{\ell}, \vec{h} \in I$
1: **if** u.Overlaps ($\vec{\ell}$, $\vec{h}$) **then**
2: u.Send_to (w, u.Get_vals ($\vec{\ell}, \vec{h}$))
3: **end if**
4: **for** $j = 0$ to $u.depth$ **do**
5: $d = u.dim[j]$ /* splitting dimension */
6: **if** $u.id[j] = 0$ and $u.split[j] < \vec{h}[d]$ **then**
7: $\vec{\ell'} = \vec{\ell}$
8: $\vec{\ell'}[d] = u.split[j]$
9: $u.link[j]$.Range ($\vec{\ell'}, \vec{h}, w$)
10: **else if** $u.id[j] = 1$ and $u.split[j] > \vec{\ell}[d]$ **then**
11: $\vec{h'} = \vec{h}$
12: $\vec{h'}[d] = u.split[j]$
13: $u.link[j]$.Range ($\vec{\ell}, \vec{h'}, w$)
14: **end if**
15: **end for**

Figure 3 illustrates an example of a range query depicted in Figure 3b issued by virtual peer u. Initially, u executes Algorithm 2 and retrieves the tuples that satisfy the range and fall in its zone (the rectangle of leaf 00). Then, u constructs the cyan part of the range as the intersection with the maximal sibling subtree rooted at depth 1. u sends this part to y (1st hop), which is u's link in that subtree; the associated message is drawn as the cyan arrow. y retrieves the tuples in the cyan area and sends them to u. Since there is no overlap of this area with any maximal sibling subtree, no further message is necessary.

In parallel, u constructs the orange part of the range, which comprises the yellow and green part, as the intersection of the query range with the maximal sibling subtree rooted at depth 2. u sends this part (1st hop) to its link v in that subtree, shown as the orange arrow in Figure 3a. Subsequently, v processes the orange range query. It extracts the yellow part that overlaps with its zone, and sends all qualifying tuples to the issuer

u. The orange range only overlaps with the maximal sibling subtree rooted at depth 2. Thus, u constructs the green part and sends it to its link w (2nd hop). Finally, retrieves the tuples in the green range and sends them to u; no further message is required.



(a) Messages sent (b) Range partitions

Fig. 3: Range query issued by u.

Note that in the example of Figure 3, the query is answered in two hops. In the first, u reaches y and v, and in the second, v reaches w. The following lemma shows that in the worst case $O(\log n)$ hops are necessary.

Lemma 4: The expected number of hops in a range query is $O(\log n)$ in the worst case.

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

Consider that virtual peer v receives from u a query for range $Q = [\vec{\ell}, \vec{h}]$; 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 maximum depth of the k-d tree. The fact that the expected maximum depth is $O(\log n)$ completes the proof. ■

The following lemma proves the correctness of Algorithm 2.

Lemma 5: Algorithm 2 returns all indexed tuples in the queried area.

Proof: Let Q denote the range and S the result set S returned to the issuer. We need to show that $\forall \vec{p} \in Q, \vec{p} \in S$. We prove by contradiction that the negation of the previous hypothesis $\exists \vec{p} \in Q \wedge \vec{p} \notin S$ is *false*.

Assume there exist a tuple $\vec{p}$ such that $\vec{p} \in Q$ but $\vec{p} \notin S$. We discern two cases:

- The virtual peer containing point $\vec{p} \in Q$ is not encountered and therefore cannot be added to S. Algorithm 2 partitions Q into a set of non-overlapping subranges that together span the entire area of Q. Hence, *all* peers whose zone overlaps the queried area Q are reached according to our routing protocol. Therefore, $\vec{p} \in Q$ is definitely reached at its owner and the assumption is false.
- Point $\vec{p} \in Q$ is not in any peer. This implies that $\vec{p}$ is "lost", which is not possible as discussed in Section III-E. Hence, this assumption is also false.

Therefore, all qualifying tuples $\vec{p} \in Q$ are included in the answer set S. ■

B. Nearest Neighbor Queries

A nearest neighbor (NN) query specifies a center $\vec{c}$ and a parameter K, and returns the K tuples nearest to the center $\vec{c}$, according to some distance metric. MIDAS can process NN queries for any distance metric defined on the D-dimensional Euclidean space I; in the following we assume Euclidean distance.

Nearest neighbor queries are more challenging than range queries, mainly because the distance of the K-th nearest neighbor from the center cannot be known in advance. This means that the extent of search cannot be restricted, as is the case with range queries. Note that a brute force method to process a NN query would be to pose a range query for the entire space and let the query issuer compile the answer set. While this approach would require $O(\log n)$ hops using the algorithm of Section IV-A, it would transfer the entire dataset to the issuer. MIDAS employs a more conservative approach that takes $O(\log^2 n)$ hops, trying to reduce the number of tuples that reach the issuer.

Initially, the query issuer w identifies the virtual peer responsible for $\vec{c}$ with a lookup query. Then, w sends a NN request message with parameters $(\vec{c}, K, M, R, H, w)$. M denotes the number of tuples already sent to w; initially $M = 0$. R is the distance from $\vec{c}$ of the farthest tuple, among the M, sent; initially $R = 0$. H is the lowest depth that the request can be forwarded to; initially $H = 0$. Intuitively, R designates the search frontier; it will slowly increase until $M = K$ tuples are found. H restricts the propagation of the request ensuring that no peer receives multiple requests.

Each virtual peer u that receives a NN request executes Algorithm 3. First, u must retrieve all its local tuples that are within distance R to the query center and insert them into set S (line 1). This is because, some of these tuples may be closer to those M already sent to the query issuer. Note that if there are more than K such tuples, S should only contain the K nearest to $\vec{c}$; any other tuple is definitely not in the result.

Currently, u has retrieved $|S|$ while the request was for $K - M$ tuples. Therefore, if $|S| < K - M$, u should expand the search frontier and retrieve up to $K - M - |S|$ additional tuples (lines 2–5). Virtual peer u retrieves as many as possible tuples closest to $\vec{c}$, inserts them into S, and sets R equal to the distance from $\vec{c}$ of the last tuple inserted.

Then, u sends all tuples in S to the issuer w (line 6) and increments M by $|S|$ (line 7), as w has received $|S|$ additional tuples.

Finally, u forwards a request to its links at depth larger than H, if necessary (lines 8–12). In particular, u considers its links starting from the one at the largest depth. This reverse traversal

of the $link$ array preserves proximity, since k-d tree leaves that reside in low subtrees (rooted at large depths) correspond to rectangles that are close to each other. A request is sent to the link (line 10) if M is still less than the desired K or if the link's zone overlaps with the search frontier, i.e., with the hypersphere of radius R centered at $\vec{c}$ (line 9).

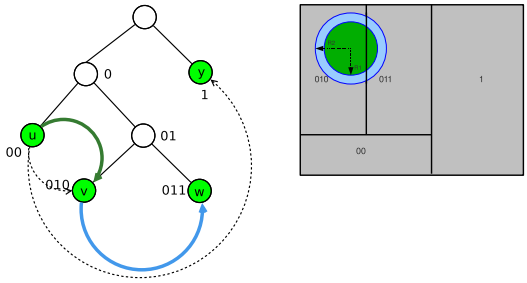
Algorithm 3 u.NN $(\vec{c}, K, M, R, H, w)$: Peer u processes a K-NN query from $\vec{c}$ issued by w; M tuples within distance R have already been sent to w; the request should not be forwarded to any link at depth lower than H

Require: $K>0$, $0\leq M<K$, $R\geq 0$, $H\geq 0$

1: retrieve up to K closest to $\vec{c}$ tuples within distance R and insert them into S
2: **if** $M+|S|<K$ **then**
3: retrieve up to $K-M-|S|$ closest to $\vec{c}$ tuples and insert them into S
4: update R to the distance of the last retrieved tuple
5: **end if**
6: u.Send_to (w, S)
7: $M = M+|S|$
8: **for** $j = u.depth$ upto H **do**
9: **if** $M<K$ or $u.link[j]$.Overlaps $(\vec{c}, R)$ **then**
10: $u.link[j]$.NN $(\vec{c}, K, M, R, j, w)$
11: **end if**
12: **end for**

Figure 4 shows a 10-NN search example issued by u. Virtual peer v is responsible for the query center and begins NN processing. Assume that v contains only 5 tuples, and thus sets $M = 5$ and $R = R_1$, shown as the green disc in Figure 4b. Then, v examines its links starting from w. Since $M = 5 < 10$, v forwards a request to w.

Subsequently, virtual peer w executes Algorithm 3. Assume that 3 tuples of w fall in the green disk. Since $5+3<10$, w needs to expand the search frontier to cover two more tuples. The result is the cyan disk with radius R_2 and M becomes 10. The search frontier does not overlap with any of w's links, and thus NN processing terminates.



(a) Messages sent (b) Search frontier

Fig. 4: A NN example

The following lemma bounds the number of hops necessary to process a NN query.

Lemma 6: Algorithm 3 returns the K-NN to query center $\vec{c}$ and the expected number of hops required is $O(\log^2 n)$ in the worst case.

Proof: We prove by contradiction. We assume that Algorithm 3 returns an answer of size less than K. In effect, when the request encounters peer u, lines 1–5 correspond to constructing an answer set S of a locally performed K-NN query.

The updated request for $K-M-|S|$ more tuples is forwarded to the non-encountered overlapping peers in $O(\log n)$ hops (range query). The algorithm does not terminate until M becomes K and all peers in distance R from $\vec{c}$ have been encountered. At that time, the issuer has received at least K tuples, as each peer gradually increases M according to the size of the answer S it returns to the issuer. Hence, the initial proposition is false and Algorithm 3 returns at least K points.

The above procedure, which includes a range search, is repeated $O(\min\{K, \log n\})$ times in the worst case. Thus, it takes at most $O(\log^2 n)$ hops for processing a K-NN query. ■

V. Experimental Evaluation

In order to assess our methods and validate our analytical results, we performed extensive experiments that simulate a dynamic environment, for each of the search types we study.

A. Setting

We evaluated performance according to several metrics appearing in the literature, related to contention mostly. We measure query performance by latency which is the maximum distance (in terms of hops) from the initial peer to any peer reached during search. Withal, we address the issue of measuring storage and query load distribution fairness. Along with latency, this constitutes a scalability measure and consequently is of major significance. Therefore, we use Jain's fairness index [19], $f(x) = \frac{(\sum_{i=1}^{N} x_i)^2}{n\sum_{i=1}^{N} x_i^2}$, where $x_i \geq 0$, to measure data- and task-skew. Yet, another metric indicative of fairness is the summing load of some top ranked percentage (eg., 10%, 20%) of peers. Albeit quite pessimistic, this metric provides an insight into how load scales for the most loaded peers of the overlay as it adapts to abrupt topology changes.

The state a node preserves is directly related to the cost of maintenance efforts during changes in the overlay network as a result of peers joining, leaving and failing. Therefore, we also study the mean state a node maintains in MIDAS relative to its competition since it impacts on scalability.

Also for each simulation cycle, the ratio between the number of accessed relevant peers to the number of peers that are relevant to the query reveals how fast the answer is retrieved. Likewise, when the ratio between the number of accessed relevant peers to the total number of encountered peers takes value 1 it is optimal, in a sense that only relevant peers were disturbed. Therefore, for each query we define the following

metrics that we record as the simulation evolves in time.

$$\text{precision}(t) = \frac{\#(\text{accessed-hosts}(t) \cap \text{relevant-hosts}(t))}{\#(\text{accessed-hosts}(t))}$$

$$\text{recall}(t) = \frac{\#(\text{accessed-hosts}(t) \cap \text{relevant-hosts}(t))}{\#(\text{total-relevant-hosts})}$$

$$\text{respons}(t) = \frac{\#(\text{reached-tuples}(t) \cap \text{relevant-tuples}(t))}{\#(\text{total-relevant-tuples})}$$

Our experiments simulate a dynamic environment. They consist of two distinct stages, a *growing* and a *shrinking* stage. Therefore, we were given the opportunity to study the course of those metrics, as overlay adapts dynamically to changes of the topology. For each cycle of the *increasing stage*, a new peer joins the overlay network, whereas during the *decreasing stage* we remove a random peer, selected at random with equal probability. In all figures we show and compare the two stages distinctly and their impact on all metrics. All simulations initiate an overlay of 1K peers, which is growing up to 100K peers one insertion at a time (growing stage), followed by the reverse procedure (shrinking stage).

For the purpose of evaluating MIDAS, we used synthetic datasets of varying skewness and dimensionality. The reason we chose synthetic instead of real datasets is the necessity to study how MIDAS and its competitors, CAN and MAAN, scale with increasing dimensionality for datasets of varying skewness. Unfortunately, real datasets with such features are not available. Henceforth, we will refer to any dataset by D_s^d, where d stands for its dimensionality degree, and s for its skewness. All dimensions are independent to each other and identically distributed. By convention, we represent a uniform distribution of keys in $[0,1]^d$ with D_0^d. Skewness factor s is a distribution function parameter of the random variable producing the data keys. More formally, we let $f_{D_s^1}(x) = (s+1)x^s$, where $x \in [0,1]$. Hence, we have that $f_{D_s^d}(\vec{x}) = \prod_{j=1}^{d} f_{D_s^1}(x_j)$, where $\vec{x} \in [0,1]^d$. All datasets consist of 1M multi-dimensional keys.

We will refer to any queryset by Q_t^d, where d stands again for the dimensionality degree of the set, whereas t stands for its type, for example range search etc. All querysets Q_t^d, have their query-centers distributed in a uniform fashion in $[0,1]^d$. A K NN query is described by a query center $\vec{c}$ and K. Likewise, for each range query we select a query center and we displace it by a random amount. Then, we create a square query around this point of random distance. One important parameter is the number of data items which will be retrieved when this query is evaluated. This configuration has been set around 50 for our experiments. All querysets consist of 50K queries, which were executed in 10 different overlay topologies in order to validate our results. In all figures we draw the mean value of all presented metrics.

An important metric in distributed systems is network *congestion*, defined as the average number of queries processed at any node, when n uniformly random queries are issued.

B. Results

In this section we present our results that validate our previous analytical claims from former sections.

1) Point Queries: Figure 5 presents query performance aspects for point queries. Latency for MIDAS is bounded by $O(\log n)$, as Lemma 3 predicts, and apparently is not affected by either dimensionality or skewness. On the other hand, the expected latency for CAN is bounded by $O(\sqrt[d]{n})$ and thus takes advantage of the increasing dimensionality. Therefore, it is only natural for CAN to outperform MIDAS for 19 dimensions. At what cost though? This is depicted in Figure 9 where the expected cardinality of a node's routing table is shown. It is confirmed to be in $O(d)$ for large CAN overlays, whereas MIDAS and MAAN are in $O(\log n)$ regardless dimensionality degree. The state a node maintains 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. Obviously, there is a tradeoff between latency and maintenance cost in CAN, which is adjusted by dimensionality degree. Nevertheless, the dimensionality degree of a dataset is fixed, and thereby, non-desirable properties might be imparted to the overlay.

CAN achieves low levels of fairness regarding data load (Figure 12), mainly due to its joining protocol. More specifically, it chooses to halve a peer's area for a newcomer; instead of splitting its data load like in MIDAS. Consequently, CAN becomes extremely valnurable to data skew. Data load fairness shows a gradual attentuation with topology changes. The reason fairness drops lies with tha nature of the join and departure procedures to convey load from one peer only, to some other peer (the split for joins or the departing peer), instead of uniformly disseminating the departing peer's load amid all peers, or assigning load from the whole overlay to a new peer in such a way that all peers are responsible for equal loads afterwards. Nonetheless, such a scheme would require coordination among all peers, which is translated to a large communication overhead.

2) Range Queries: We now turn our attention to the performance of our range search method. We span all values of the range query simultaneously and thereby reaches in parallel the relevant peers. Therefore, latency shows similar behavior with our exact search method and has a logarithmic behavior, as Lemma 4 predicts. Figure 6 exhibits MIDAS' conspicuous lineaments to be unaffected by dimensionality. In addition, skewness has insignificant impact on latency. On the other hand, latency on MAAN is dominated by the number of peers relevant to the query (Figure 8), due to the adopted approach to look up for a bound of the range first, and then sequentially traverse all relevant neighboring peers.

Furthermore, dimensionality ameliorates precision. This is quite similar to the effect that the overlay size has on precision. Precision is improved by increasing the size of the overlay, due to the fact that peers become responsible for smaller areas of responsibility, while our queries have fixed size. Therefore, granularity is enhanced and less irrelevant peers are reached. Likewise, task-load fairness improves with dimensionality

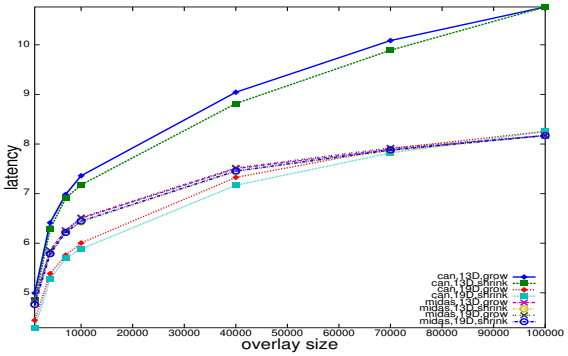


Fig. 5: Latency for point queries.

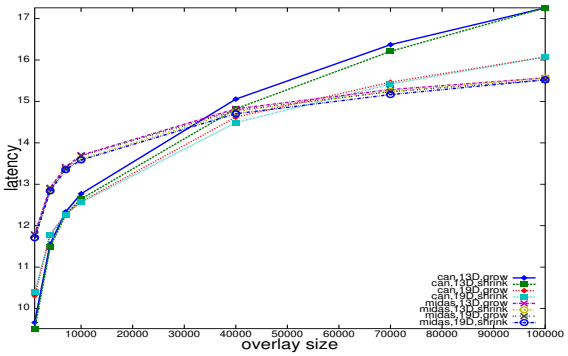


Fig. 6: Latency for range queries.

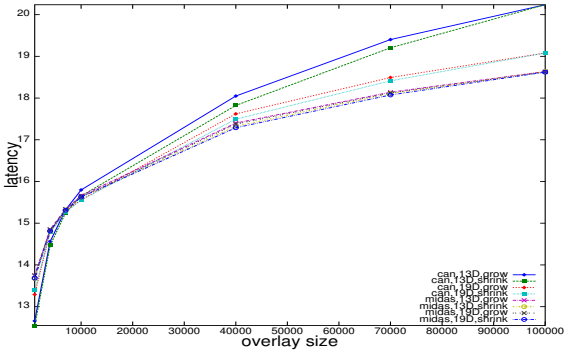


Fig. 7: Latency for 50-NN search queries.

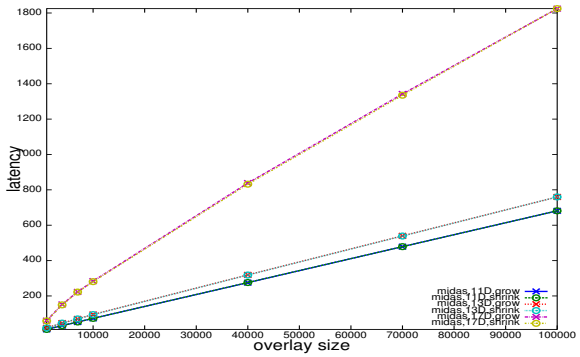


Fig. 8: Latency for range queries in MAAN.

degree (Figure 13). Finally, Figure 10 shows the congestion for range queries. For low data skewness, congestion has a logarithmic behavior [21] as depicted in the figure.

3) K Nearest Neighbor Queries: In Figure 7 we compare MIDAS with the method presented in [20] for CAN. This approach takes advantage of the vicinity in the construction of CAN's routing tables in order to disseminate a query to the peers surrounding its center. Unlike our previous methods, latency of a KNN search is bounded by $O(\log^2 n)$, as Lemma 6 predicts. Note that Figure 11 shows the congestion for nearest neighbor queries.

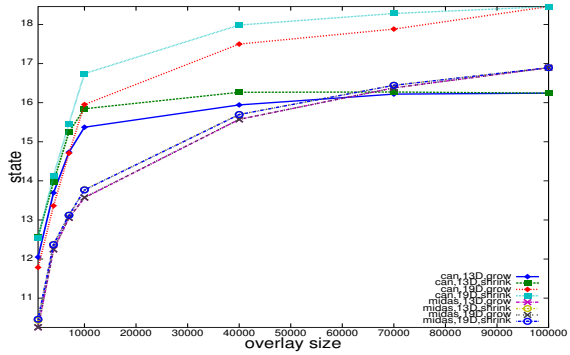


Fig. 9: State a peer maintains for D_1^d.

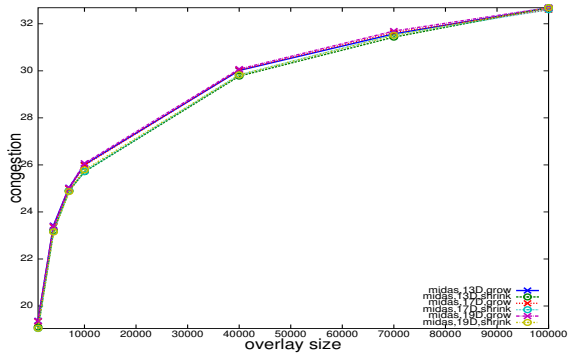


Fig. 10: Congestion for range queries.

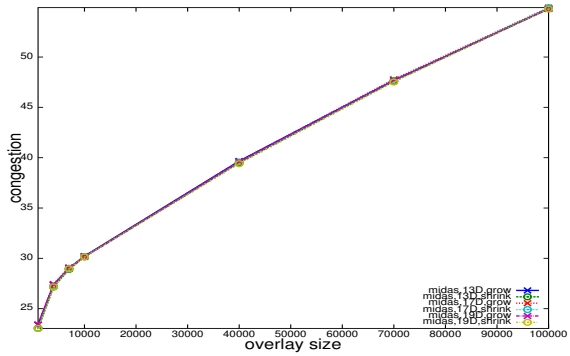


Fig. 11: Congestion for 50-NN search queries.

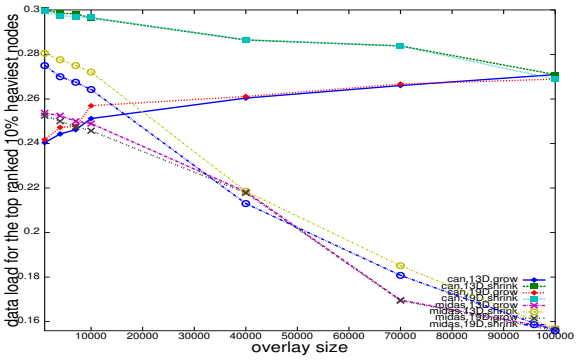


Fig. 12: Data load fairness index for D_1^d.

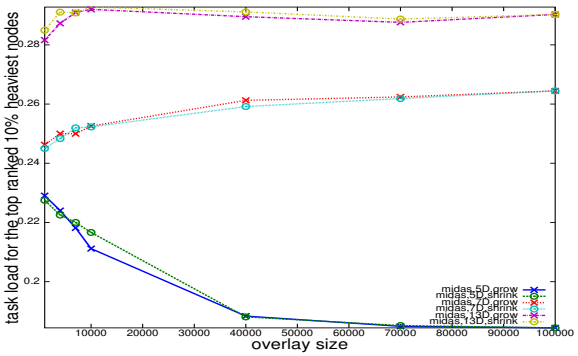


Fig. 13: Summed task-load fairness index of the top 10% ranked nodes for range search (Q^1_{rng} in D_1^d)

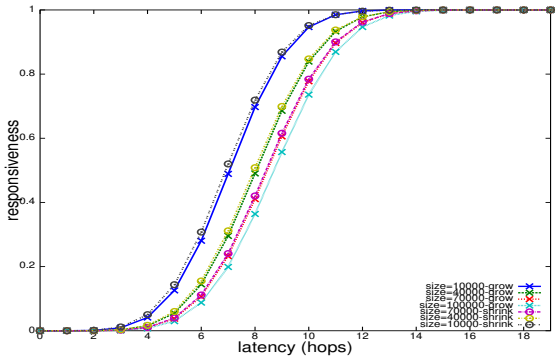


Fig. 14: Responsiveness for range search queries.

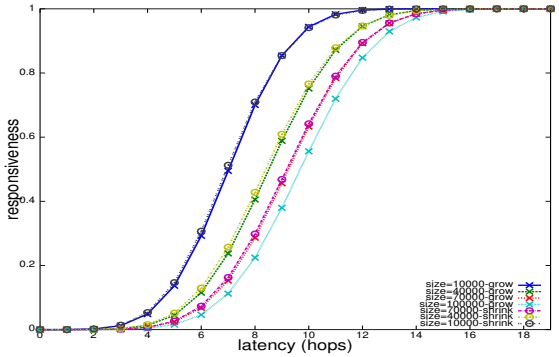


Fig. 15: Responsiveness for 50-NN search queries.

VI. Conclusion

In this work we have presented MIDAS, a pure multi-dimensional 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 multi-dimensional structures in terms of scalability. MIDAS allows for multidimensional queries and offers guarantees concerning all operations. In particular, updates, lookups and range queries are resolved in $O(\log n)$ hops, whereas K-NN queries in $O(\log^2 n)$ 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] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: supporting scalable multi-attribute range queries," in *SIGCOMM*, 2004, pp. 353–366.

[2] M. Cai, M. R. Frank, J. Chen, and P. A. Szekely, "Maan: A multi-attribute addressable network for grid information services," *J. Grid Comp.*, vol. 2, no. 1, pp. 3–14, 2004.

[3] P. Ganesan, B. Yang, and H. Garcia-Molina, "One torus to rule them all: Multidimensional queries in p2p systems," in *WebDB*, 2004, pp. 19–24.

[4] 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*, 2005, pp. 173–180.

[5] S. Blanas and V. Samoladas, "Contention-based performance evaluation of multidimensional range search in p2p networks," in *InfoScale'07*, 2007, pp. 1–8.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01*, 2001, pp. 161–172.

[7] H. V. Jagadish, B. C. Ooi, and Q. H. Vu, "Baton: A balanced tree structure for peer-to-peer networks," in *VLDB*, 2005, pp. 661–672.

[8] 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*, 2006, p. 34.

[9] C. G. Plaxton, R. Rajaraman, and A. W. Richa, "Accessing nearby copies of replicated objects in a distributed environment," *Theory Comput. Syst.*, vol. 32, no. 3, pp. 241–280, 1999.

[10] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Softw.*, vol. 3, no. 3, pp. 209–226, 1977.

[11] 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.*, 1997, pp. 654–663.

[12] 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.*, vol. 11, no. 1, pp. 17–32, 2003.

[13] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth,

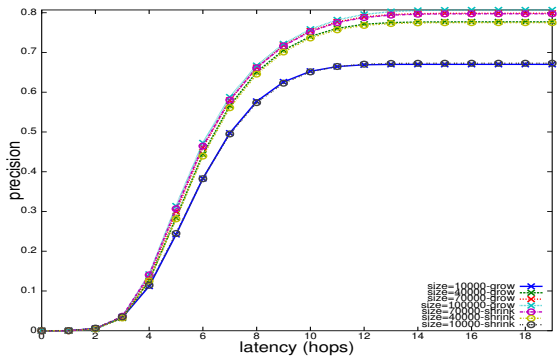


Fig. 16: Precision for range queries in D_1^2.

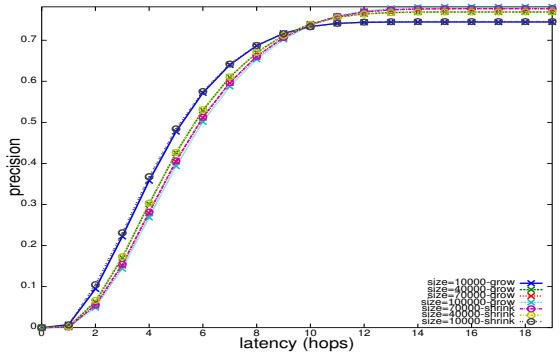


Fig. 17: Precision for range queries in D_1^{13}.

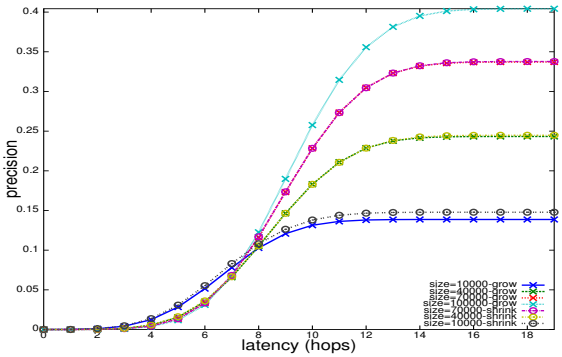


Fig. 18: Precision for 50-NN search queries.

M. Punceva, and R. Schmidt, "P-grid: a self-organizing structured p2p system," *SIGMOD Record*, vol. 32, no. 3, pp. 29–33, 2003.

[14] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *IPTPS*, 2002, pp. 53–65.

[15] B. Zhao, J. Kubiatowicz, and A. D. Joseph, "Tapestry: a resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Comm.*, vol. 22, no. 1, pp. 41–53, 2004.

[16] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware*, 2001, pp. 329–350.

[17] J. L. Bentley, "K-d trees for semidynamic point sets," in *Symposium on Computational Geometry*, 1990, pp. 187–197.

[18] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer, "Range queries in trie-structured overlays," in *P2P Computing*, 2005, pp. 57–66.

[19] R. Jain, D. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," in *DEC Research Report TR-301*, 1984.

[20] F. Falchi, C. Gennaro, and P. Zezula, "Nearest neighbor search in metric spaces through content-addressable networks," *Inf. Process. Manage.*, vol. 44, no. 1, pp. 411–429, 2008.

[21] M. Argyriou, V. Samoladas, and S. Blanas, "Grasp: generalized range search in p2p networks," in *InfoScale '08*, 2008, pp. 1–10.