Geoinformatica
DOI 10.1007/s10707-012-0163-x

Index-based query processing on distributed multidimensional data

George Tsatsanifos · Dimitris Sacharidis · Timos Sellis

Received: 31 January 2012 / Revised: 29 June 2012 /
Accepted: 12 July 2012

Abstract This work introduces decentralized query processing techniques based on MIDAS, a novel distributed multidimensional index. In particular, MIDAS implements a distributed k-d tree, where leaves correspond to peers, and internal nodes dictate message routing. MIDAS requires that peers maintain little network information, and features mechanisms that support fault tolerance and load balancing. The proposed algorithms process point and range queries over the multidimensional indexed space in only $O(\log n)$ hops in expectance, where n is the network size. For nearest neighbor queries, two processing alternatives are discussed. The first, termed eager processing, has low latency (expected value of $O(\log n)$ hops) but may involve a large number of peers. The second, termed iterative processing, has higher latency (expected value of $O(\log^2 n)$ hops) but involves far fewer peers. A detailed experimental evaluation demonstrates that our query processing techniques outperform existing methods for settings involving real spatial data as well as in the case of high dimensional synthetic data.

Keywords Distributed index · Distributed query processing · Peer-to-peer networks · Nearest neighbor query

G. Tsatsanifos · T. Sellis
National Technical University of Athens, Athens, Greece

G. Tsatsanifos
e-mail: gtsat@dblab.ece.ntua.gr

D. Sacharidis (✉) · T. Sellis
Institute for the Management of Information Systems,
R.C. "Athena", Athens, Greece
e-mail: dsachar@imis.athena-innovation.gr

T. Sellis
e-mail: timos@imis.athena-innovation.gr

1 Introduction

There is a continuous interest to improve information retrieval in decentralized networks of machines, or peers. The most efficient techniques follow the structured network paradigm, and find application in lookup services, resource discovery mechanisms, and content delivery systems, among others. In structured networks, a global protocol mandates data distribution over peers, which essentially translates searching for a piece of information into locating the peer responsible for it.

Even though a multitude of structured networks have been proposed, only a few of them are capable of storing and querying multidimensional data. We briefly discuss existing work by distinguishing three categories. The first includes solutions that directly extend some single-dimensional structured network. 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., [4, 6]. However, these approaches still have to resort to only one of the dimensions for processing queries. The most popular approach, [5, 10, 20], 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 structured 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 structured systems that were explicitly designed to store multidimensional information, e.g., [10, 17]. The basic idea is that each peer is responsible for a rectangular region of the space, and has knowledge of its neighbors in adjacent regions. Being multidimensional in nature, their processing cost for most queries is sublinear to the network size. 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 nodes, i.e., jumps cannot be made.

The last category includes methods, e.g., [11, 12], 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; e.g., the experiments in [12] showed that for dimensionality close to 20, this method was outperformed by the non-indexed approach of [17].

Motivated by these observations, this work proposes query processing techniques based on a novel structured network, called Multi-attribute Indexing for Distributed Architecture Systems (MIDAS) [22]. MIDAS takes a different approach than existing methods. 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 [16] for single dimensional tree-like structures. Unlike other multidimensional index techniques, e.g., [12], 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.

Using the MIDAS infrastructure, the most important classes of multi-attribute queries can be efficiently processed in arbitrary dimensionality. In particular, we show that, for a network of n peers, point queries and range queries are processed in $O(\log n)$ hops, in expectance. These bounds are smaller than non-indexed d-dimensional structured systems, e.g., $O(d\sqrt[d]{n})$ of [17]. Furthermore, we propose two processing alternatives for nearest neighbor search. The first, termed eager processing, has low latency (expected value of $O(\log n)$ hops) but may involve a large number of peers. The second, termed iterative processing, has higher latency (expected value of $O(\log^2 n)$ hops) but is more conservative and involves far fewer peers than eager processing. A thorough experimental study on real spatial data as well as on synthetic data of varying dimensionality validates these claims.

The remainder of this paper is organized as follows. Section 2 describes MIDAS and its basic operations including load balancing and fault tolerance mechanisms. Section 3 discusses multidimensional query processing. Section 4 presents an extensive experimental evaluation of our processing techniques. Section 5 reviews related work, and Section 6 concludes this work.

2 MIDAS architecture

This section presents the information stored in each peer and details the basic operations in the MIDAS overlay network. In particular, Section 2.1 introduces the distributed index structure, Section 2.2 discusses the information stored within each peer in MIDAS. Section 2.4, 2.3 and 2.5 elaborates on the actions taken when a peer departs, joins, and fails, respectively. Section 2.6 discusses load balancing and fault tolerance.

2.1 Index structure

The distributed index of MIDAS is an instance of an adaptive k-d tree [3]. Consider a D-dimensional space $I = [\mathbf{l}_I, \mathbf{h}_I]$, defined by a low $\mathbf{l}_I$ and a high $\mathbf{h}_I$ 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 2.3. 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 $I_{2i} = [\mathbf{l}_{2i}, \mathbf{h}_{2i}]$, $I_{2i+1} = [\mathbf{l}_{2i+1}, \mathbf{h}_{2i+1}]$. Assuming that the left child ($T[2i]$) is assigned the lower part of I_i, it holds that (1) $\mathbf{l}_{2i}[d_j] = \mathbf{l}_{2i+1}[d_j]$ and $\mathbf{h}_{2i}[d_j] = \mathbf{h}_{2i+1}[d_j]$ on every dimension $d_j \neq d_i$, and (2) $\mathbf{h}_{2i}[d_i] = \mathbf{l}_{2i+1}[d_i] = s_i$ 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.

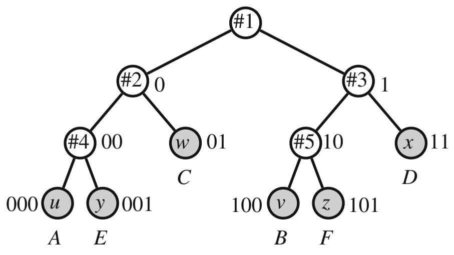
Each node of the k-d tree is associated with a binary identifier corresponding to its path from the root, which is defined recursively. The root has the empty id $\varnothing$; the left (resp. right) child of an internal node has the id of its parent augmented with 0 (resp. 1). Figure 1a depicts a k-d 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 Fig. 1a; the splits are numbered and shown next to the corresponding axis parallel cuts.

In this work, a tuple is a key-value pair, where the *key* is D-dimensional. Thus, a key 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 whose keys fall in its rectangle. The hierarchical structure of the k-d tree allows for efficient methods to process queries, such as range and nearest neighbor queries.

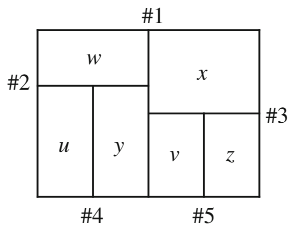
2.2 MIDAS peers

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 2.4, 2.5), or for load balancing and fault tolerance purposes (Section 2.6).

A peer in MIDAS corresponds to a leaf of the k-d tree, and stores/indexes all key-value tuples, whose keys reside in the leaf's rectangle, which is called its *zone*. A peer is denoted with small letters, e.g., u, v, w, whereas a physical peer with capital letters, e.g., A, B, C. For example, in Fig. 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 Fig. 1a, do not correspond to peers and of course not to physical peers. An important property of peers in MIDAS is the following *invariant*.



(a) A k-d tree



(b) Leaf rectangles

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

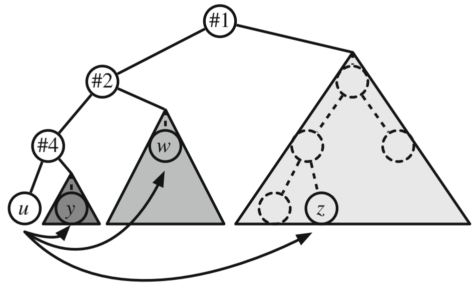
Lemma 1 *For any point in space I, there exists exactly one peer in MIDAS responsible for it.*

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. □

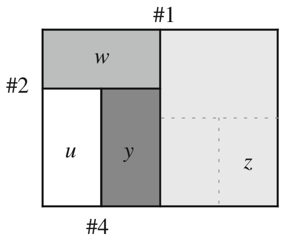
A peer u in MIDAS contains only partial information about the k-d tree, which however is sufficient to perform complex query processing discussed in Section 3. In particular, peer u contains the following state. (1) $u.id$ is a bitmap representing the leaf's binary id; $u.id[j]$ is the j-th most significant 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. (7) $u.address$ is the network address of the physical peer responsible for u.

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 Fig. 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.

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, Fig. 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



(a) Maximal sibling subtrees of u



(b) Local knowledge at u

Fig. 2 Links of peer u

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 Fig. 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 3.1, and process queries, as discussed in Section 3.

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.

2.3 Peer joins

When a new physical peer joins MIDAS, it becomes responsible for a single peer. Initially, the newly arrived physical peer chooses a random point p in the space I and locates the peer v responsible for it (Section 3.1 explains the lookup process). 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 randomly 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 $w.backlink$. (6) w notifies its backlinks about its address $w.address$.

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 new address.

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: $v(100)$	

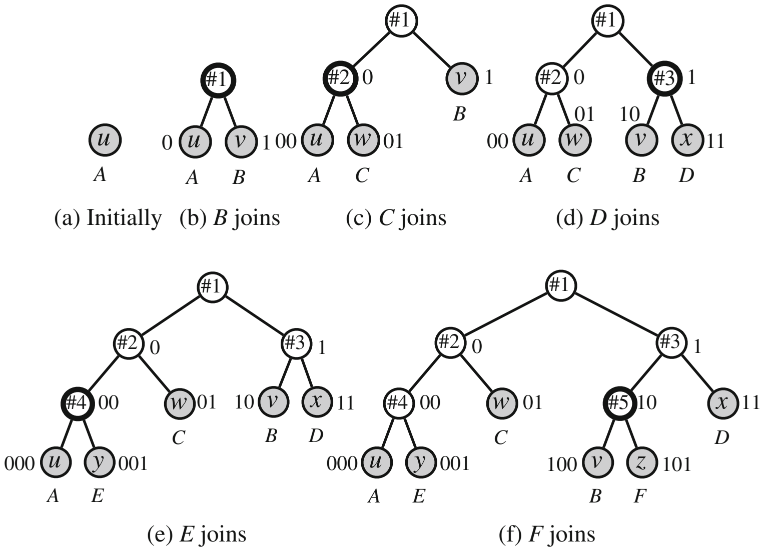


Fig. 3 Network creation

We present an example of how the network of Fig. 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 Fig. 3a. Then, physical peer B joins and causes a split of the k-d tree root along the first dimension (Split #1 in Fig. 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.

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 Fig. 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 Fig. 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 [2, 8] 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 [18] 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 3.1 and 3, the theorem provides bounds for the cost of query processing.

2.4 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 Fig. 1a, assume that physical peer E, responsible for y, departs. Peer y's sibling is 000, which is a leaf and

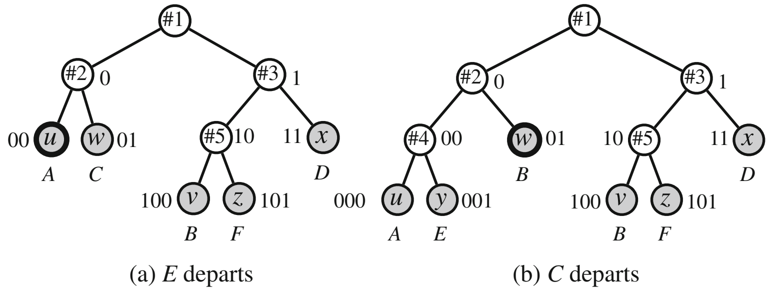


Fig. 4 The two scenarios for peer departures

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 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[1] (see also Section 2.6). Note that the backlinks of w must be notified about the address of the new peer responsible for w. In the example of Fig. 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 4a 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.

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

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. □

2.5 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 tuples 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 tuples 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 2.6.

2.6 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 [21]. 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 tuples are refreshed (see Section 2.5), the physical peer can pose the query to another reality.

3 Query processing on MIDAS

This section details multi-attribute query processing on MIDAS. In particular, Section 3.1 discusses point queries, Section 3.2 range queries, and Section 3.3 nearest neighbor queries.

3.1 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 $\mathbf{q}$. If its zone contains $\mathbf{q}$, it returns the tuples with key $\mathbf{q}$ (if they exist) 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 $\mathbf{q}$. 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 $\mathbf{q}$ falls in (lines 5–11). The query is then forwarded to the link corresponding to that subtree (line 7).

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

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

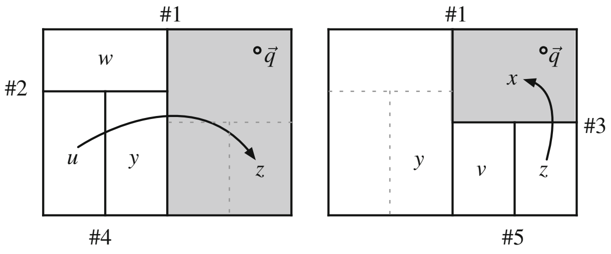
To illustrate the previous procedure, consider a query for point $\mathbf{q}$ issued by peer u. Figure 5a draws $\mathbf{q}$ and the local knowledge about the space I at peer u. Observe that $\mathbf{q}$ falls outside u's zone. Peer u thus forwards the query to its link z within the shaded area since it contains $\mathbf{q}$ (1st hop). Next, peer z processes the query. Point $\mathbf{q}$ is inside the shaded area of Fig. 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 $\mathbf{q}$ 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.

Consider that peer v receives from u a query for the requested point $\mathbf{q}$; assume that v is the link at depth k in the *link* array of u. Due to how the algorithm (line 7) forwards the query, $\mathbf{q}$ is contained within the subtree that contains v rooted at depth

Fig. 5 Point query example for **q**



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

k. Therefore, **q** cannot be contained in 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.

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. □

3.2 Range queries

A range query specifies a rectangular area Q in the space, defined by a lower **l** and a higher point **h**, and requests all tuples that fall in Q. Instead of locating the peer responsible for a corner of the area, e.g., **l**, 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 $Q = [\mathbf{l}, \mathbf{h}]$ 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 j.

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 Fig. 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 Fig. 6a. Then, u constructs the shaded regions, shown in Fig. 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 Fig. 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 the range with the areas in its maximal sibling subtrees. Observe that

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

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

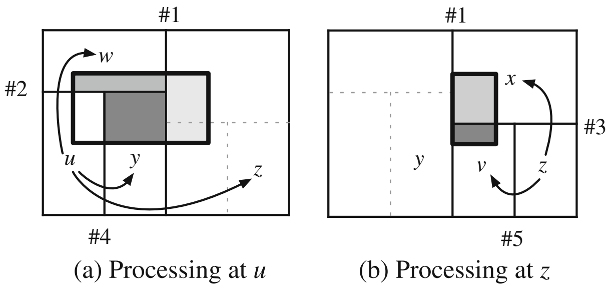


Fig. 6 Range query example

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 Fig. 6b (2nd hop). Finally, peers v and x process the queries locally as the requested ranges have no overlap with their zones.

As explained in the example of Fig. 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 peer v receives from u a query for range $Q = [\mathbf{l}, \mathbf{h}]$; 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, range query processing requires as many rounds, or hops, as the depth of the k-d tree. The fact that the expected depth is $O(\log n)$ (from Theorem 1) completes the proof. □

3.3 Nearest neighbor queries

Given a center **c** and a parameter K, a nearest neighbor (NN) query requests the K tuples nearest to the center **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. Therefore, 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 only require $O(\log n)$ hops using the algorithm of Section 3.2, it would transfer the entire dataset to the issuer. In the following, we present two approaches that are significantly more efficient. The first, termed *eager processing*, requires $O(\log n)$ hops and in practice retrieves much fewer tuples than the brute force method. The second, termed *iterative processing*, tries to further reduce the number of retrieved tuples and takes $O(\log^2 n)$ hops. Note that while both algorithms try to minimize the number of retrieved tuples, they may be forced to retrieve the entire dataset in extreme situations, e.g., when K is very large (in the order of the dataset size).

3.3.1 Eager processing

The goal of this approach is to compute, in a distributed manner, an upper bound of the distance of the K-th closest to the center tuple. Similar to range query processing, MIDAS poses a request to multiple peers in parallel. Each request is accompanied by a guarantee (M, R), which means that M tuples within distance R from **c** are already retrieved. Eager processing concludes, i.e., MIDAS stops forwarding requests, when M becomes greater than K *and* all tuples within distance R from **c** are retrieved. When this occurs, R is not smaller than the distance of the K-th closest **c** tuple, and the issuer has retrieved a superset of the K-NN result set.

Let w denote the peer that issued the K-NN query. Initially, w locates the peer, say z, responsible for **c** with a lookup query. We refer to peer z as the *manager*. All peers that receive a request during eager processing, including the manager, execute Algorithm 3.

A request specifies seven parameters: the center **c** and the value of K, the guarantee (M, R), the depth threshold D, and the addresses of the issuer w and the manager z. The role of the threshold D is to restrict propagation of the request within the subtree rooted at depth D, ensuring that no peer receives multiple requests. Note that the manager is the first peer to receive a request with $M = 0$, $R = 0$, and $D = 0$, initially. Intuitively, the role of a request is to notify a peer about the current guarantee (M, R), and ask for additional tuples. Based on its local knowledge, the peer: (1) retrieves and transmits some tuples to the issuer; (2) updates the guarantee to (M', R'), where $M' \geq \min\{M, K\}$, while R' can be smaller, equal or greater than R; and (3) sends new requests with the updated guarantee.

Algorithm 3 u.NN ($\mathbf{c}$, K, M, R, D, w, z)
Peer u processes a K-NN query at center $\mathbf{c}$ issued by w and managed by z.
M tuples within distance R are already known.
The request should only be forwarded to links within the subtree at depth D.

```
1: insert in $S$ up to $K$ closest to $\mathbf{c}$ tuples within distance $R$
2: if $|S| = K$ then                                        ▷ case I
3:     $R \leftarrow r(S)$
4:     $M \leftarrow K$
5: else if $M + |S| \geq K$ then                            ▷ case II
6:     $M \leftarrow M + |S|$
7: else if $M + |S| < K$ then                               ▷ case III
8:     insert in $S$ up to $K - M - |S|$ closest to $\mathbf{c}$ tuples
9:     $R \leftarrow \max\{R, r(S)\}$
10:    $M \leftarrow M + |S|$
11: end if
12: $u$.Send_to ($w$, $S$)
13: $u$.Send_to ($z$, $|S|$, $r(S)$)               ▷ only for iterative processing
14: for $j \leftarrow D + 1$ to $u.depth$ do
15:     if $M < K$ or $u.link[j]$.Overlaps ($\mathbf{c}$, $R$) then
16:         $u.link[j]$.NN ($\mathbf{c}$, $K$, $M$, $R$, $j$, $w$, $z$)
17:     end if
18: end for
```

Consider a peer u that receives a request and executes Algorithm 3. First, u retrieves the local tuples within distance R to the query center, and inserts them into set S (line 1). Note that the peer will eventually transmit S to the issuer (line 12). Therefore, it is necessary to include all those tuples within distance R, as some of them could be part of the K-NN result set. Of course, if there are more than K such tuples, only the K nearest to the center should be inserted in S. There is no need to insert any farther tuple as it definitely cannot be in the result.

Next, the peer constructs a new guarantee, based on the set S and the received guarantee (M, R); if necessary, the peer will retrieve additional tuples. Depending on the number of tuples inserted in S, three cases exist. In the first, S contains exactly K tuples (lines 2–4); recall that it cannot contain more. Let $r(S)$ denote the distance from $\mathbf{c}$ of the farthest tuple in S. Peer u can make the guarantee $(K, r(S))$, i.e., K tuples within distance $r(S)$ exist. Observe that this is a stronger guarantee than the one received (M, R), since K tuples are known to exist within a smaller distance $r(S) \leq R$. As a result M and R are updated with the values of $r(S)$ and K, respectively (lines 3, 4).

In the second case (lines 5–6), there are less than K but not less than $K - M$ tuples in S, i.e., $M + |S| \geq K$. All of them are within distance R to the center. Therefore, peer u can guarantee that within distance R, there exist additionally $|S|$ tuples for a total of $M + |S|$ tuples. Therefore, it updates the value of M to $M + |S|$ (line 6).

In the third case (lines 7–11), the number of tuples in S is less than $K - M$, i.e., $M + |S| < K$. In this situation, the peer can only make a guarantee for less than K tuples. However, it is possible that local tuples at distance greater than R exist. Therefore, the peer retrieves at most $K - M - |S|$ additional tuples closest to the center (line 8), and the values of R and M are updated to reflect the new guarantee (lines 9, 10).

At this point (M, R) represents the new guarantee that peer u can provide. Note that the new value of M is either K (first case) or not smaller than the original (second and third cases). On the other hand, the new value of R can be smaller (first case), equal (second case), or greater (third case) than the original.

Subsequently, all tuples in S are sent to the issuer w (line 12). Please note that line 13 applies only to iterative processing (see Section 3.3.2) and is thus skipped. The final step (lines 14–18) is sending a request to other links. Observe that a request is only sent to links that are in a sibling subtree rooted at depth greater than D (line 14); the reason for this restriction is explained shortly. A request is sent to one of u's links (line 16), if more tuples are required, i.e., $M < K$, or if the link may contain tuples within distance R from $\mathbf{c}$ (line 15). The request will have the same parameters as the one received by u, with the exception that M, R represent the new guarantee, and the depth threshold is set to the link's depth j.

In the previous process, when peer u sends a request to its j-th link, say v, it sets the depth threshold equal to j. This way, u ensures that the request sent to v can only be further propagated within u's maximal sibling tree rooted at depth j. Note that such a restriction is not necessary for range query processing, as the range is partitioned into non-overlapping parts in each request.

Figure 7 illustrates eager processing for a 2-NN query on our running example. The white circles represent the tuples in the network, while the cross represents the query center $\mathbf{c}$. Assume that the query is issued on peer u, whose zone covers $\mathbf{c}$. Initially, u executes Algorithm 3 for $M = R = 0$. Set S remains empty after executing line 1. Then, the third case applies. As only one tuple exists in u's zone (drawn as a filled circle in Fig. 7a), $M = 1$ and R becomes equal to the radius of the shaded disc shown in Fig. 7a. Subsequently, since $M < K$, peer u sends a request message to all its links, y, w, and z, indicated by the arrows in Fig. 7a, with D set to 3, 2, and 1, respectively.

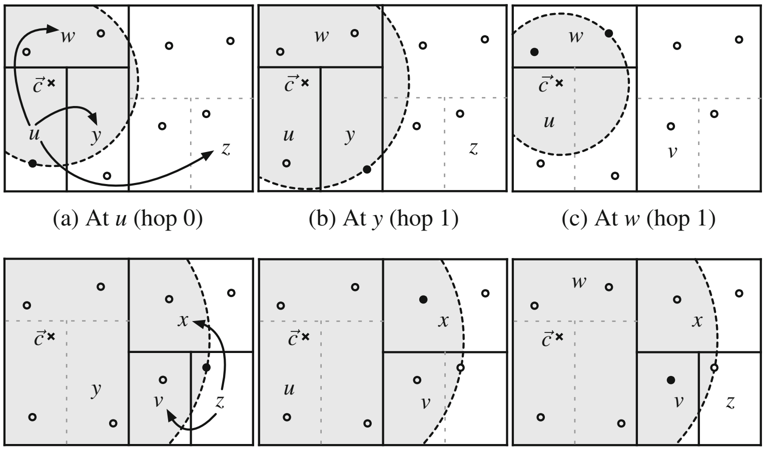


Fig. 7 2-NN eager processing

Next, consider processing at peer y, which is 1 hop away from the issuer u. Observe that y has no tuples within range R, depicted as the shaded disc of Fig. 7a. Therefore, the third case applies: y retrieves its only tuple (the filled circle in Fig. 7b), increases M by 1, and sets R to the radius of the larger shaded disc shown in Fig. 7b. Peer y does not send a new request, as parameter $D = 3$ is equal to its depth.

In parallel, processing at peer w takes place. There exist two tuples of w (the filled circles in Fig. 7c) within the shaded disc of Fig. 7a, and hence the first case applies ($|S| = 2 = K$). M becomes two, while R is shrunk to the distance of the farthest from $\mathbf{c}$ tuple in S. This distance defines the shaded disc depicted in Fig. 7c. Peer w does not send a new request, as threshold $D = 2$ is equal to its depth.

Figure 7d shows processing at peer z, which is 1 hop away from the issuer. The third case applies, as the zone of z does not intersect the disc of Fig. 7a. Therefore, z retrieves its only tuple (the filled circle in Fig. 7d), sets $M = 2$ and increases R to the radius of the shaded disc shown in Fig. 7d. Since $D = 1$ and $z.depth = 3$, peer z sends a request with the updated (M, R) to its links at depth 2 and 3, i.e., peers x and v, respectively.

Processing at x and v occurs in parallel at the second hop, and is illustrated in Fig. 7e and f. In both peers, the second case of Algorithm 3 applies, as $M = 2$. Since there is a tuple within the disc of Fig. 7d, M increases by 1, while R is not changed (the disk remains the same in Figs. 7e and f). Peers x and v do not send any requests as their threshold D is equal to their depths.

Upon conclusion of the distributed eager processing, the issuer u has received the seven tuples drawn as filled circles in Fig. 7. Among them, u chooses the two closest to $\mathbf{c}$, which happen to be those of peer w.

The next two lemmas show the complexity of eager processing and prove its correctness.

Lemma 6 *The expected number of hops for eagerly processing a nearest neighbor query is* $O(\log n)$.

Proof Initially, the issuer locates the manager by issuing a point query that requires $O(\log n)$ hops. Then, the manager and all other peers execute Algorithm 3. Upon receipt of a request with D parameter value, a peer will only send new requests to peers within the subtree rooted at depth D. Moreover, all these new requests have increased D values (see lines 14–18 in Algorithm 3). Therefore, in each hop, requests are propagated to smaller subtrees. In the worst case, the number of hops necessary is equal to the depth of the k-d tree. The fact that the expected depth is $O(\log n)$ (from Theorem 1) completes the proof. □

Lemma 7 *Eager processing correctly identifies the* K *nearest neighbors to center* $\mathbf{c}$.

Proof We prove by contradiction that the issuer receives a superset of the K nearest neighbors. Assume that there exists a tuple, say τ, which is among the K nearest neighbors but is not sent to the issuer. This may happen in one of two scenarios, depending on whether the peer, say u, responsible for τ is reached.

In the first scenario, u is reached, i.e., executes Algorithm 3 but does not include $\boldsymbol{\tau}$ in S. Certainly, $\boldsymbol{\tau}$ is not within distance R. Case I (lines 2–4) suggests that u has K tuples which are closer to $\mathbf{c}$ than $\boldsymbol{\tau}$ — a contradiction. Case II (lines 5–6) suggests that there exist more than K tuples ($M + |S| \geq K$) within distance R. Similarly, case III (lines 7–11) suggests that there exist K tuples closer than $\boldsymbol{\tau}$ (since not all tuples of u, e.g., $\boldsymbol{\tau}$, are included in S, we have $M + |S| = K$). In cases II and III, this can only be true if the received guarantee (M, R) is not valid, i.e., there do not exist at least M tuples within distance R.

Therefore, it suffices to show that, given a valid guarantee, Algorithm 3 constructs a new valid guarantee. This clearly holds in case I. In cases II and III, it also holds because M is incremented by local tuples and R is correctly updated to the distance of the farthest local tuple included or retains its value. In any case the new guarantee is valid, which implies that the first scenario is impossible.

In the second scenario, u is not reached. This may happen in two cases. Either the condition in line 15 does not hold for the subtree that contains u, or that subtree is never considered. The condition does not hold if $M \geq K$ *and* the subtree of u is not within distance R, which implies that $\boldsymbol{\tau}$ cannot be a K nearest neighbor (we have shown that all guarantees are valid). The other case, i.e., that no subtree containing u is considered, never occurs. This is because the issuer initially sets D to 0, and all peers consider all sibling subtrees at depth up to D. □

3.3.2 Iterative processing

Similar to eager processing, iterative processing computes an upper bound of the distance of the K-th closest tuple to the center. However, this method involves a series of rounds orchestrated by the manager. The basic idea is to direct the search towards peers that are close to the center and progressively expand to more remote ones, if necessary. Specifically, in each round, the manager restricts requests to propagate only within a subtree containing the center. Starting with the smallest possible subtree, the manager iteratively enlarges it until either the K-NN query is answered or the entire k-d tree is reached.

Initially, the query issuer w locates the manager z (i.e., the peer responsible for the query center $\mathbf{c}$), which executes Algorithm 4. On the other hand, upon receiving a request, all other peers execute Algorithm 3 discussed in the previous section. Note that in addition to sending the local tuples retrieved (set S) to the issuer, the peer also sends a *local guarantee* $(|S|, r(S))$ to the manager (line 13 of Algorithm 3). This essentially summarizes the contribution of the peer towards the K-NN answer. Based on all local guarantees collected in a round, the manager decides whether another round is necessary.

We next elaborate on the role of the manager and discuss Algorithm 4. First, the manager z retrieves up to K closest to the center tuples stored locally (line 1), and sends them to the issuer (line 2). Furthermore, it composes a local guarantee for these tuples and inserts it into set G (line 3), which stores all local guarantees received. Subsequently, execution proceeds in rounds (lines 4–19). Starting from z's depth, the value of the depth threshold D is decreased by one in each round.

At the beginning of each round, a global guarantee (M, R) that summarizes all locals is constructed (lines 5–11). In particular, the local guarantees in G are sorted according to the distance R (line 5) and global M is initialized to zero (line 6). Then, the manager examines them in sequence (lines 7–11). The local guarantee (M_i, R_i) updates the global (M, R) (lines 8, 9). If M becomes greater than K, then R is definitely greater or at least equal to the distance of the K-th nearest neighbor. In

Algorithm 4 z.NN_manage ($\mathbf{c}$, K, w)
Peer z manages a K-NN query at center $\mathbf{c}$ issued by w.

```
 1: insert in $S$ up to $K$ closest to $\mathbf{c}$ tuples
 2: $z$.Send_to ($w$, $S$)
 3: insert in $G$ local guarantee ($|S|$, $r(S)$)
 4: for $D \leftarrow z.depth$ down to 1 do
 5:     sort entries in $G$ according to distance
 6:     $M \leftarrow 0$
 7:     for each $(M_i, R_i) \in G$ do
 8:         $M \leftarrow M + M_i$
 9:         $R \leftarrow R_i$
10:         if $M \geq K$ then break
11:     end for
12:     if $M < K$ or $z.link[D]$.Overlaps ($\mathbf{c}$, $R$) then
13:         $z.link[D]$.NN ($\mathbf{c}$, $K$, $M$, $R$, $D$, $w$, $z$)
14:         repeat
15:             $z$.Receive ($M_i$, $R_i$)
16:             insert in $G$ local guarantee ($M_i$, $R_i$)
17:         until messages from $D$ hops away are received
18:     end if
19: end for
```

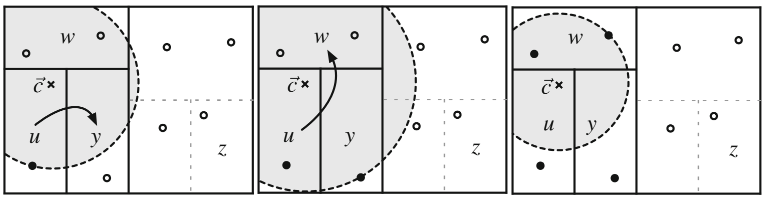
this case, (M, R) is the strongest global guarantee that can be derived from G. Hence, there is no need to examine other local guarantees (line 10).

After this process, (M, R) summarizes the knowledge obtained from the local guarantees. The manager sends a request to its link at depth D only if more tuples are required, or the link may contain tuples with distance R from the center (lines 12, 13). Note that the depth threshold of the request is set to D. As a result, the link propagates the request within its subtree at depth D. Then, the manager receives local guarantees from all peers reached in this round, i.e., it waits until a message from up to D hops away is received (lines 14–17).[2] The local guarantees received are inserted in the set G and are thus used in the computation of the global guarantee in the following round.

Figure 8 illustrates the role of the manager in iterative processing for the same 2-NN query discussed in the previous section. Initially, the manager, peer u in this example, retrieves the single tuple within its zone; let R_u denote its distance to the center $\mathbf{c}$. Hence, in the first round ($D = u.depth = 3$) the global guarantee is $(1, R_u)$ and is depicted as a disc in Fig. 8a. Since $1 < K$, the manager sends a request to its link y at depth 3. Then, peer y executes Algorithm 3 and retrieves the only tuple in its zone at distance R_y (see Fig. 7b). Also it sends to the manager the corresponding local guarantee $(1, R_y)$.

At the beginning of the second round ($D = 2$), the manager has two local guarantees: its own, $(1, R_u)$, and that of y, $(1, R_y)$. After executing lines 5–11, it

[2] In our implementation, a timeout process is initiated. Note that missed messages do not affect the algorithm's correctness, since the global guarantee is correctly computed (see proof of Lemma 9) on the set G of retrieved local guarantees.



(a) Round 1 (depth 3) (b) Round 2 (depth 2) (c) Round 3 (depth 1)

Fig. 8 2-NN iterative processing at manager u

obtains the global guarantee $(2, R_y)$, shown as a disc in Fig. 8b. The manager then sends a request to its link w at depth 2, since the region of the subtree rooted at that depth overlaps the disc. Peer w retrieves two tuples and sends a local guarantee $(2, R_w)$, where R_w is the distance to $\mathbf{c}$ of the farthest tuple (see Fig. 7c).

In the third round ($D = 1$), the manager computes the global guarantee $(2, R_w)$ depicted in Fig. 8c, by taking into account only w's local guarantee, since $R_w < R_u < R_y$. Note that the disc does not overlap with the region of the subtree rooted at depth 1, and thus no further request is necessary.

Upon conclusion of the iterative processing, the issuer has received the four tuples drawn as filled circles in Fig. 8. Compare this to the seven tuples sent in eager processing.

The next two lemmas show the complexity of iterative processing and prove its correctness.

Lemma 8 *The expected number of hops for iteratively processing a nearest neighbor query is $O(\log^2 n)$.*

Proof Initially, the issuer locates the manager by issuing a point query that requires $O(\log n)$ hops. Then, the manager executes Algorithm 4. The maximum number of required rounds is equal to the depth of the managing peer. In each round, processing is confined in a subtree of the k-d tree, and every contacted peer executes Algorithm 3. Following the proof of Lemma 6, a round concludes in number of hops equal to the subtree depth, in the worst case. From Theorem 1, the depth of the k-d tree, and therefore of any peer and subtree, is $O(\log n)$ in expectance. As a result the expected number of hops in iterative processing is $O(\log^2 n)$. □

Lemma 9 *Iterative processing correctly identifies the K nearest neighbors to center* $\mathbf{c}$.

Proof The proof is similar to that of Lemma 7 with two differences. First, we need to show that the global guarantees constructed by the manager are valid. Note that all received local guarantees are valid (see proof of Lemma 7). Then it is straightforward to see that Lines 5–11 correctly compute a global guarantee from the set of local guarantees.

Second, let u denote a peer that contains a nearest neighbor, but assume u is not reached. This may happen in two cases. Either the subtree of the manager that contains u is not considered, or the condition in line 11 of Algorithm 4 does not hold for this subtree. The former cannot occur as the manager examines all subtrees (D is iteratively decreased down to 1). The latter can only occur if the global guarantee is invalid — a contradiction. □

4 Experimental study

In order to assess our methods and validate our analytical results, we simulate a dynamic environment and evaluate network characteristics and query performance.

4.1 Setting

Methods We compare our approach MIDAS against three popular techniques from literature. MAAN [6] belongs to the category of single-dimensional structured network that are extended for multi-attribute data. On the other hand, CAN [17] is an inherently multi-dimensional structured network. Finally, the VBI-tree [12] is the current state-of-the-art method among decentralized hierarchical multi-dimensional indices.

Network We simulate a dynamic topology that captures arbitrary physical peer joins and departures, in two distinct stages. In the *increasing stage*, physical peers continuously join the network while no physical peer departs. It starts from a network of 1,000 physical peers and ends at 100,000 physical peers. On the other hand, in the *decreasing stage*, physical peers continuously leave the network while no new physical peer joins. This stage starts from a network of 100,000 physical peers and ends when only 1,000 physical peers are left. When we vary the network size, the figures show the results during the increasing stage; the results during the decreasing stage are analogous and omitted.

Data and queries We use real and synthetic datasets. The real dataset, obtained from the R-Tree Portal,[3] denoted as *NE*, consists of spatial locations representing approximately 125K postal addresses in three metropolitan areas, New York, Philadelphia and Boston. The synthetic dataset, denoted as *Synthetic*, contains 1M random tuples, where each attribute value is an independent identically distributed random variable X, with the probability density function $f_{X=x} = 2x$ for $0 \leq x \leq 1$, and $f_{X=x} = 0$ otherwise.

For a point query, we choose uniformly and independently a random tuple from the dataset. For a range query, we also choose a random tuple as the start of the rectangular range, while the size of the range is set so that the query returns a specific number of tuples, termed the selectivity. Similarly, the center of a nearest neighbor query is a random tuple.

Parameters Our experimental evaluation examines three parameters. The *network size* is varied from 1,000 up to 100,000 physical peers. The *dimensionality* of the *Synthetic* dataset is varied from 2 up to 13. Finally, the *selectivity*, i.e., the number of qualifying tuples in a range query or the number of neighbors in a NN query,

[3]http://www.rtreeportal.org

Table 2 System parameters

Parameter	Default	Values
Network size	10K	1K, 4K, 7K, 10K, 40K, 70K, 100K
Dimensionality	2	2, 3, 5, 7, 11, 13
Selectivity	50	25, 50, 75, 100, 125, 150

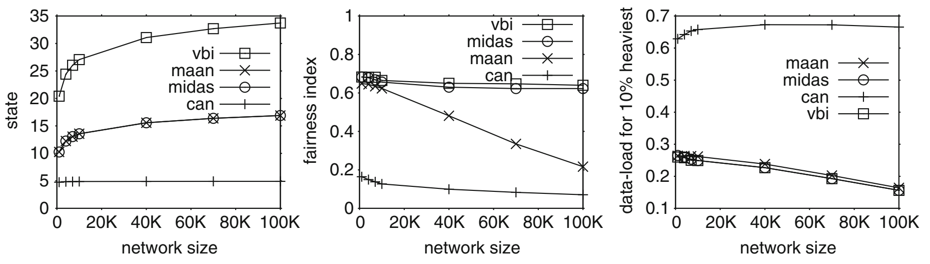
is varied from 25 up to 150 tuples. The tested ranges and default values for these parameters are summarized in Table 2. When we vary one parameter, the other two are set at their default values. Every reported value is the average of executing 50,000 queries over 15 distinct networks.

Metrics For all indexing methods, we measure their overhead on physical peers, as well as their query performance. Regarding network overhead, we measure the amount of information (e.g., links, zone, etc.), termed *state*, a physical peer must maintain. As the number of physical peers increases, this is an important measure of scalability. Moreover, we study the allocation of data among physical peers. Depending on the dataset distribution and network topology, large imbalances can occur. *Data load* measures the percentage of the total data that resides in the Q % heaviest physical peers. Its optimal value is $Q/100$, and corresponds to the condition where data is evenly allocated among all physical peers. We also use Jain's *fairness index* [13] as another metric for data allocation among physical peers. The fairness index is normalized in [0, 1], and its optimal value is 1.

Regarding query processing performance, we employ three metrics. *Latency* measures the number of hops required during processing; lower values suggest faster response. Distributed query processing imposes a load on multiple physical peers, including ones that may not contribute to the answer. Two metrics quantify this load. *Precision* is defined as the ratio of the number of physical peers that contribute to the answer over the total number of physical 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 physical peer, when n uniformly random queries are issued (n is the network size); lower values suggest lower load.

4.2 Evaluation

Network overhead Figures 9a and 10a depict the average state a physical peer maintains, as we vary the network size for the real, and the dimensionality for the synthetic dataset, respectively. State is an important metric as it is directly related



(a) State (b) Fairness index (c) Data load on heaviest 10%

Fig. 9 Network metrics for NE

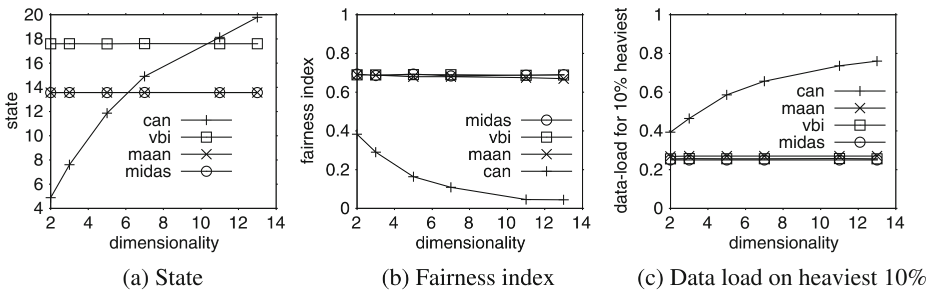


Fig. 10 Network metrics for Synthetic

to the amount of traffic that occurs due to maintenance operations like detecting failures, preserving updated routing tables each time a physical peer joins or leaves. The state in MIDAS, MAAN, and the VBI-tree increases logarithmically with the network size as seen in Fig. 9a. While, MIDAS and MAAN have comparable state, the VBI-tree requires around twice as much state. The reason is that a physical peer in the VBI-tree keeps information not only about the physical peers on its path to the root, but also about physical peers at the same depth of the tree. Furthermore, note that the state of MIDAS, MAAN, and the VBI-tree is independent of the dimensionality, as depicted in Fig. 10a. On the other hand, the state of CAN is independent of the network size, but grows linearly with dimensionality. For low dimensional datasets, such such as the spatial dataset *NE* in Fig. 9a, each CAN physical peer has few neighbors. However, while the dimensionality increases, so does the state of a CAN physical peer. For example, in 8-dimensional datasets CAN requires more state than MIDAS and MAAN.

Figure 9b and c show the data allocation metrics, fairness index and data load, on the 10 % heaviest physical peers, for the real dataset, as the network size varies. Similarly, Fig. 10b and c present these metrics as the dimensionality of the synthetic dataset varies. The main conclusions here are the following. Both the VBI-tree and MIDAS are equally fair, and are robust both to network size and data dimensionality changes. On the other hand, CAN is considerably unfair. For example, Fig. 10c shows that the 10 % heaviest physical peers in CAN contain more than 75 % of the entire 13-dimensional dataset. Finally, MAAN is fair only for small network sizes; as the number of physical peers increases, the single dimensional approach of MAAN forces unequal distribution of data, illustrated in Fig. 9b.

Point queries Figures 11 and 12 illustrate point query performance on the real dataset as the network size varies, and on the synthetic dataset as the dimensionality varies, respectively. The latency of MIDAS, MAAN and the VBI-tree scales logarithmically with the network size (Fig. 11a) and is independent of the dimensionality (Fig. 12a). In all examined cases, MIDAS and MAAN have the lowest latency and outperform the VBI-tree. Regarding CAN, its expected latency $O(\sqrt[d]{n})$ increases with the network size, but decreases with dimensionality, as depicted in the two figures. Recall that as the dimensionality increases, the state of CAN also increases (Fig. 10a).

With respect to the congestion metric, depicted in Figs. 11b and 12b, MIDAS causes the lowest congestion for all network and all dimensionalities up to 10. When

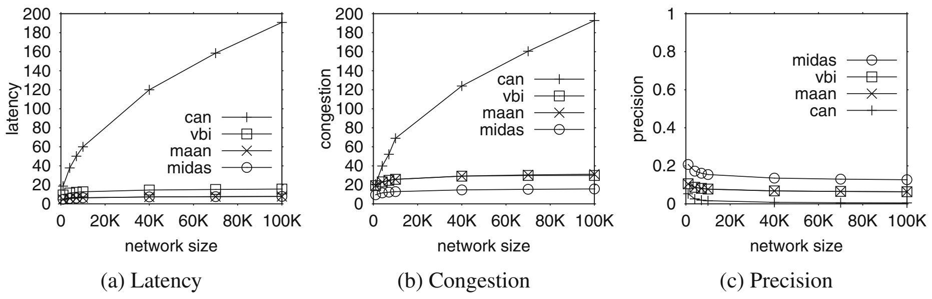


Fig. 11 Point query processing vs. network size for NE

the dataset has more than 10 attributes, CAN begins to outpeform MIDAS. On the other hand, CAN's congestion scales badly as the number of physical peers increases. The VBI-tree is the second best method with respect to congestion. Similar results hold for the precision metric, shown in Figs. 11c and 12c; note that higher precision values are preferred. Regarding MAAN, observe that its congestion (resp. precision) is low (resp. high) for spatial datasets, but grows rapidly (resp. remains constant) in larger dimensionalities. This phenomenon is due to the single-dimensional indexing of MAAN, which forces MAAN to contact multiple times the set of physical peers necessary to process the query.

Range queries The latency for range query processing of MIDAS, the VBI-tree, and CAN exhibits similar trends with the case of point query processing, as shown in Figs. 13a and 14a. In particular, MIDAS latency for processing range queries scales logarithmically with respect to network size and is unaffected by dimensionality. MAAN's latency is contingent on its single-dimensional index; it worsens as the network size or dimensionality increases.

For all methods the congestion metric worsens as the network size (Fig. 13b) or dimensionality (Fig. 14b) increases. Note that the congestion of MIDAS, the VBI-tree, and MAAN for range queries increases much faster than CAN with respect to network size compared to the case of point queries. Regarding precision, Figs. 13c

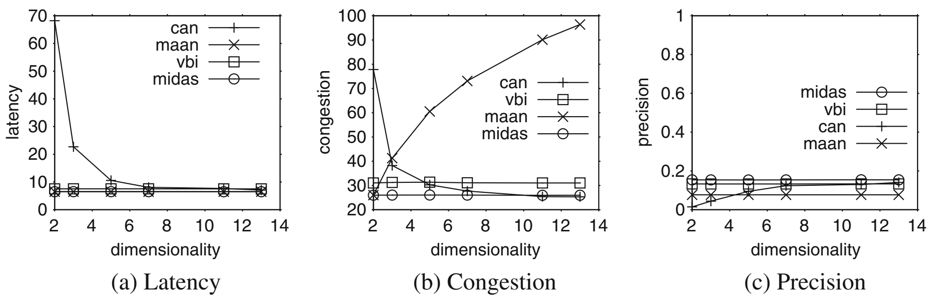


Fig. 12 Point query processing vs. dimensionality for synthetic

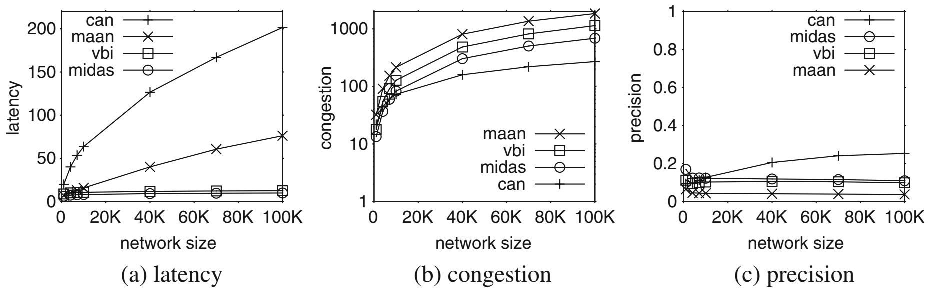


Fig. 13 Range query processing vs. network size for NE

and 14c show that the precision of all methods improves as the network size or dimensionality increases, with the exception of CAN whose precision improves with the number of peers.

Figure 15 examines the behavior of all methods, as the selectivity, i.e., the number of qualifying tuples in the range query, is varied. Latency is unaffected by selectivity for all multi-dimensional approaches, and increases linearly for the single-dimensional MAAN. In terms of the network load, the congestion in all methods worsens as the number of returned tuples increases. On the other hand, the precision improves slightly for all methods.

Figures 13b, 15b and 14b illustrate that the congestion in MIDAS is relatively low when compared to its competitors. MIDAS' congestion can be explained by the fact that it spans all values of the range query in parallel. Hence, multiple parallel routes are used simultaneously in order to fulfill a single range query. However, VBI-tree peers are more congested at all cases due to additional horizontal routes in the tree. This is a prudent approach for the VBI-tree, as horizontal traversals aim at alleviating the peers corresponding to the root and the other top nodes of the tree, which otherwise would become a critical bottleneck. Nonetheless, this effort causes an overhead with respect to latency and congestion. Besides, for MIDAS, internal nodes do not correspond to actual peers, but constitute routing directives instead, and therefore, none such consideration is needed. Moreover, in Figs. 13c

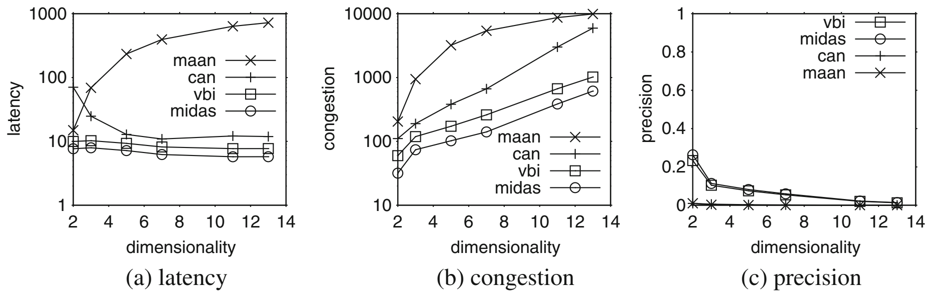


Fig. 14 Range query processing vs. dimensionality size for synthetic

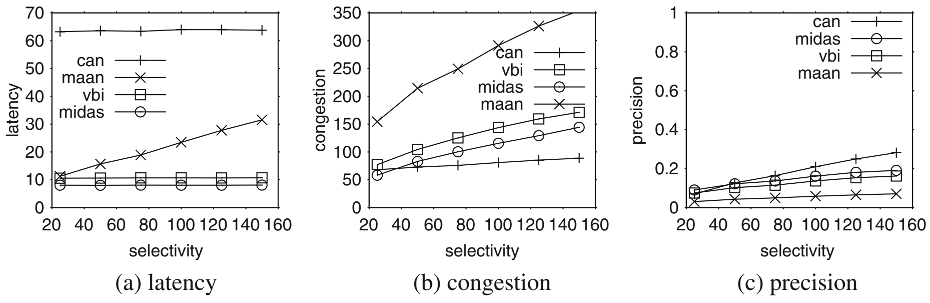


Fig. 15 Range query processing vs. selectivity for NE

and 14c, precision is worse for VBI-tree and MAAN compared to MIDAS because they encounter more peers until all relevant peers are accessed. In Fig. 15c precision improves with query size, as the number of encountered relevant peers grows faster than the overall accessed peers.

Nearest neighbor queries Regarding nearest neighbor search, MIDAS' latency for K-NN iterative processing search is bounded by $O(\log^2 n)$, and by $O(\log n)$ for eager processing, as Lemmas 6 and 8 predict, respectively, according to a worst-case analysis. However, in Fig. 16a both methods show logarithmic behavior with respect to the overlay size. So does the VBI-tree, unlike MAAN's latency which increases linearly with overlay size. For CAN, we compare MIDAS with the method presented in [9]. This approach takes advantage of the vicinity in the construction of CAN's routing tables in order to disseminate a query to the physical peers surrounding its center. Albeit effective, CAN performs poorly for low-dimensional workloads, like the spatial NE workload, as shown in Fig. 16a.

Figure 16b illustrates congestion with respect to the overlay size. Quite notably, the average load per physical peer for MIDAS iterative processing is the lowest, a result which is accompanied by high levels of precision in Fig. 16c. Our eager processing method, despite having low latency, involves a significant burden on the

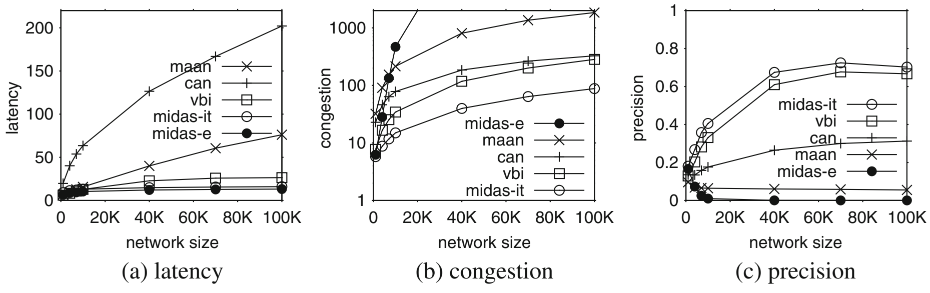


Fig. 16 50-NN query processing vs. network size for NE

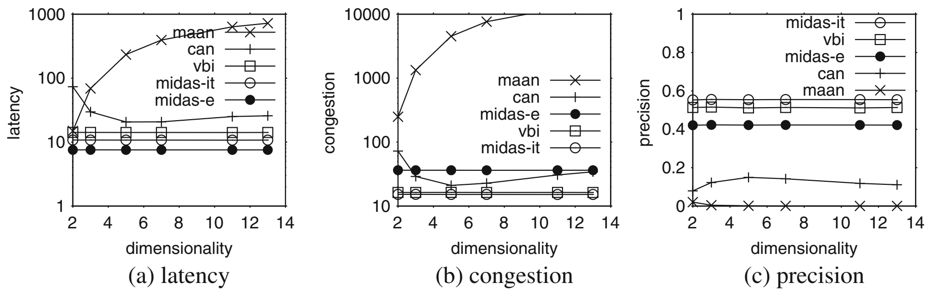


Fig. 17 50-NN query processing vs. dimensionality for synthetic

network due to the bad initial estimate of the radius of the query. Additionally, this estimate worsens as the network becomes bigger in Fig. 16c.

In Fig. 17 both MIDAS methods for nearest neighbor search are unaffected by the increased dimensionality and are asymptotically better than all competitors. On the other hand, CAN exploits dimensionality by establishing a number of links per physical peer analogous to the dimensionality, and therewith, latency and congestion ameliorate drastically. MAAN suffers severely from the curse of dimensionality as latency, congestion and precision worsen dramatically.

Finally, selectivity, depicted in Fig. 18, affects MIDAS iterative processing slightly, as it exhibits the best latency, congestion and precision. Eager processing, has also constant latency, but its congestion and precision scale badly with selectivity. CAN is also not affected by selectivity, but has much larger values. The latency of VBI-tree and MAAN slightly increases with selectivity. Figure 18b shows that the congestion of all methods increases with selectivity, with CAN, the VBI-tree and iterative processing being the most robust. Figure 18c shows that precision improves with selectivity for CAN, VBI-tree and iterative processing method, as the number of encountered relevant physical peers grows faster than the overall accessed physical peers. On the other hand, precision deteriorates for eager processing.

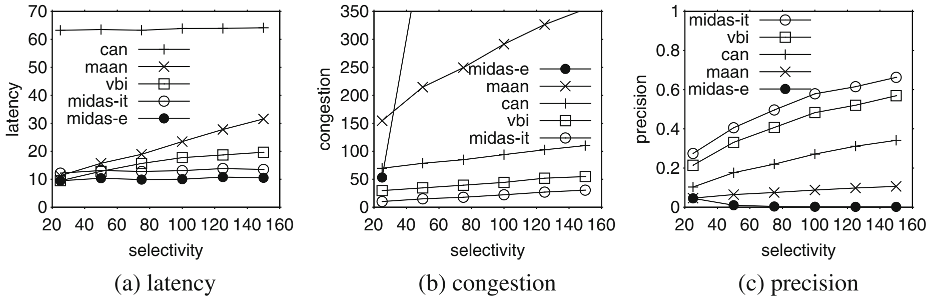


Fig. 18 50-NN query processing vs. selectivity for NE

5 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 [14] 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 [21] 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 [15], Tapestry [24] and Pastry [19]. Peer lookup in these systems is based on Plaxton's algorithm [16]. 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 [17], 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. Note that an R-Tree based adaptation of CAN for cloud computing environments is described in [23]. Analogous results hold for MURK [10], 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 [10], ZNet [20], 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 [12] 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.

6 Conclusion

This work presented query processing techniques for decentralized networks using a multidimensional indexing scheme termed MIDAS. MIDAS differs significantly from other popular systems, and allows for multidimensional queries with latency guarantees. In particular, point and range queries are, in expectance, resolved in $O(\log n)$ hops, where n is the size of the network. For nearest neighbor searching, MIDAS offers two alternatives. The first has low latency (expected value of $O(\log n)$ hops) but may involve a large number of peers. The second has higher latency (expected value of $O(\log^2 n)$ hops) but involves far fewer peers. A detailed experimental evaluation demonstrated that MIDAS outperforms existing methods, in real and synthetic datasets, while introducing low burden on peers. An interesting future direction would be to investigate the implementation of MIDAS in a cloud computing environment, similarly to how CAN was extended in [23].

References

1. Aberer K, Cudré-Mauroux P, Datta A, Despotovic Z, Hauswirth M, Punceva M, Schmidt R (2003) P-grid: a self-organizing structured p2p system. SIGMOD Record 32(3):29–33
2. Bentley JL (1975) Multidimensional binary search trees used for associative searching. Commun ACM 18(9):509–517
3. Bentley JL (1990) K-d trees for semidynamic point sets. In: Symposium on computational geometry, pp 187–197
4. Bharambe AR, Agrawal M, Seshan S (2004) Mercury: supporting scalable multi-attribute range queries. In: SIGCOMM, pp 353–366
5. Blanas S, Samoladas V (2007) Contention-based performance evaluation of multidimensional range search in p2p networks. In: InfoScale'07, pp 1–8
6. Cai M, Frank MR, Chen J, Szekely PA (2004) Maan: a multi-attribute addressable network for grid information services. J Grid Comp 2(1):3–14
7. Datta A, Hauswirth M, John R, Schmidt R, Aberer K (2005) Range queries in trie-structured overlays. In: P2P Computing, pp 57–66
8. Duch A, Estivill-Castro V, Martínez C (1998) Randomized k-dimensional binary search trees. In: ISAAC, pp 199–208
9. Falchi F, Gennaro C, Zezula P (2008) Nearest neighbor search in metric spaces through content-addressable networks. Inf Process Manag 44(1):411–429
10. Ganesan P, Yang B, Garcia-Molina H (2004) One torus to rule them all: multidimensional queries in p2p systems. In: WebDB, pp 19–24
11. Jagadish HV, Ooi BC, Vu QH (2005) Baton: a balanced tree structure for peer-to-peer networks. In: VLDB, pp. 661–672

12. Jagadish HV, Ooi BC, Vu QH, Zhang R, Zhou A (2006) Vbi-tree: a peer-to-peer framework for supporting multi-dimensional indexing schemes. In: ICDE, p 34
13. Jain R, Chiu D, Hawe W (1984) A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. In: DEC Research Report TR-301
14. Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D (1997) Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In: ACM Symp. on Theory of Comp., pp 654–663
15. Maymounkov P, Mazières D (2002) Kademlia: a peer-to-peer information system based on the xor metric. In: IPTPS, pp 53–65
16. Plaxton CG, Rajaraman R, Richa AW (1999) Accessing nearby copies of replicated objects in a distributed environment. Theory Comput Syst 32(3):241–280
17. Ratnasamy S, Francis P, Handley M, Karp R, Schenker S (2001) A scalable content-addressable network. In: SIGCOMM '01, pp 161–172
18. Reed BA (2003) The height of a random binary search tree. J ACM 50(3):306–332
19. Rowstron AIT, Druschel P (2001) Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware, pp 329–350
20. Shu Y, Ooi BC, Tan KL, Zhou A (2005) Supporting multi-dimensional range queries in peer-to-peer systems. In: Peer-to-Peer computing, pp 173–180
21. Stoica I, Morris R, Liben-Nowell D, Karger DR, Kaashoek MF, Dabek F, Balakrishnan H (2003) Chord: a scalable p2p lookup protocol for internet applications. IEEE/ACM Trans Netw 11(1):17–32
22. Tsatsanifos G, Sacharidis D, Sellis T (2011) Midas: multi-attribute indexing for distributed architecture systems. In: Proceedings of the international symposium on spatial and temporal databases (SSTD)
23. Wang J, Wu S, Gao H, Li J, Ooi BC (2010) Indexing multi-dimensional data in a cloud system. In: SIGMOD, pp 591–602
24. Zhao B, Kubiatowicz J, Joseph AD (2004) Tapestry: a resilient global-scale overlay for service deployment. IEEE J Sel Areas Commun 22(1):41–53

George Tsatsanifos is a PhD Student at the School of Electrical and Computer Engineering of the National Technical University of Athens, Greece. He received his Diploma degree from the Department of Electronic and Computer Engineering of the Technical University of Crete. His research interests include indexing, spatial databases, distributed systems, peer-to-peer systems, RDF stores.

Dimitris Sacharidis is a Postdoctoral Fellow at the Institute for the Management of Information Systems (IMIS) of Research Center "Athena", Greece. He received his BSc and PhD degrees from the National Technical University of Athens (NTUA), and his MSc degree from the University of Southern California. His research interests include spatio-temporal databases, indexing techniques, ranking, data stream algorithms, and privacy.

Timos Sellis is the Director of the Institute for the Management of Information Systems (IMIS) of Research Center "Athena", Greece and a Professor at the National Technical University of Athens (NTUA), Greece. He received his Diploma from NTUA, his MSc degree from Harvard University, and his PhD from the University of California at Berkeley, where he was a member of the INGRES group. He was an Associate Professor at the Department of Computer Science of the University of Maryland, College Park. He has received the Presidential Young Investigator award for 1990–1995 and the VLDB 1997 10 Year Paper Award for his work on spatial databases. He was the president of the National Council for Research and Technology of Greece (2001–2003) and a member of the VLDB Endowment (1996–2000). His research interests include data streams, peer-to-peer databases, data warehouses, the integration of Web and databases, and spatio-temporal databases.