V. March and Y. M. Teo, *Multi-Attribute Range Queries on Read-only DHT*, Proceedings of 15th International Conference on Computer Communications and Networks, pp. 419–424, IEEE Communications Society Press, USA, October 2006.

Multi-Attribute Range Queries on Read-Only DHT

Verdi March* and Yong Meng Teo*†
*Department of Computer Science, National University of Singapore
†Singapore-MIT Alliance, National University of Singapore
Email: [verdimar,teoym]@comp.nus.edu.sg

***Abstract*— R-DHT is a class of DHT whereby each node supports "read-only" accesses to its key-value pairs, but does not allow key-value pairs belonging to other nodes to be written on it. Recently, supporting efficient multi-attribute range queries on DHT has been an active area of research. This paper presents the design and evaluation of Midas, an approach to support multi-attribute range queries on R-DHT. Midas indexes multi-attribute resources using a d-to-one mapping scheme, and optimizes a range query by searching only for available keys. Our simulation results show that Midas on R-DHT achieves a higher lookup resiliency than conventional DHT, and it has a lower cost of query processing when the query selectivity is much larger than the number of query results.**

I. Introduction

Distributed hash table (DHT), as with a hash-table data structure, provides interfaces to store and lookup a key-value pair. DHT offers lookups with high result guarantee and short path length, even when looking for rare key-value pairs[1] [1]. To achieve this objective, DHT maps each key to a node, and organizes nodes as a structured overlay network. A lookup request for a key is routed to the *responsible node* of the key, i.e. the node where the key is mapped onto.

R-DHT is a read-only DHT where each node supports "read-only" accesses to its key-value pairs, but does not allow key-value pairs belonging to other nodes to be written (mapped) on it [2]. In contrast, conventional DHT stores key-value pairs belonging to one node on another node, which results in key-value pairs being distributed across the overlay network. The main features of R-DHT are as follows. Firstly, R-DHT supports node autonomy in placing key-value pairs [3]–[6] such that every node stores only its own key-value pairs[2]. This addresses potential soft issues such as privacy, ownership, administration and accountability of key-value pairs. Secondly, R-DHT supports high resiliency without requiring replication because when a node fails, only its own key-value pairs become unavailable. Thirdly, R-DHT prevents stale key-value pairs when the key-value pairs are updated. Lastly, R-DHT supports the flat naming scheme whereas an approach such as SkipNet [4] uses the hierarchical naming scheme to achieve controlled placement of key-value pairs.

The lookup operation of R-DHT supports efficient exact queries, i.e. resources matches exactly the *search key*. Recently, a number of approaches have been proposed to support d-attribute range queries using DHT. An example of 2-attribute range queries is to find a compute resource whose `cpu` = $P3$ and 1 GB $\leq$ `memory` $\leq$ 2 GB. However, current approaches such as distributed inverted index [8]–[12], d-to-d mapping scheme [13]–[15], and d-to-one mapping scheme [14]–[18], assume that the underlying DHT distributes key-value pairs across the overlay network.

This paper proposes Midas (**m**ult**i**-**d**imensional r**a**nge querie**s**), an approach to support multi-attribute range queries on R-DHT. Midas consists of two parts: *indexing* and *query processing*. Based on a d-to-one mapping scheme, Midas indexes a d-attribute resource by assigning it an m-bit key. To process a range query, Midas transforms the query into a number of search keys using a d-to-one mapping function. Then, it performs DHT lookups only for *available keys*, i.e. search keys that represent available resources.

To evaluate the performance of multi-attribute range queries on R-DHT, we compare the performance of two Midas implementations: one on Chord [19] and another Chord-based R-DHT (subsequently referred as R-Chord). Our simulation evaluation shows that, firstly, R-Chord achieves a higher lookup resiliency even if a fraction of nodes fail simultaneously. Secondly, the query cost in R-Chord is affected by the number of available keys whereas in Chord, it is affected by the number of search keys. Thirdly, despite the perceived overhead of R-Chord virtualization under churn, R-Chord-based Midas retrieves nearly as many keys as Chord-based Midas.

The rest of this paper is organized as follows. Section II discusses the related work. Section III presents the design of Midas. Section IV presents the experimental analysis. Section V concludes this paper.

II. Related Work

Based on the indexing scheme, we classify multi-attribute range query on DHT as distributed inverted index, d-to-d mapping, and d-to-one mapping.

Distributed inverted index such as MAAN [8], CANDy [9], Harren et. al. [10], KSS [11], and MLP [12], assigns d keys to each d-attribute resource. Typically, a locality-preserving hash function is used to hash similar attributes into similar keys [8], [20]. Two main strategies are used in the query processing. The first strategy requires d DHT lookups; one lookup per attribute. The intermediate result sets of these lookups are intersected to produce a final result set, either at the query initiator [8] or by pipelining the intermediate result sets through a number of nodes [9], [11], [12]. The second strategy requires only one lookup by taking only one intermediate result set to derive

[1]Key-value pairs that are not highly replicated and seldomly requested.
[2]This is also known as the "pay-for-your-own" model [7].

the query answers, but requires each key-value pair to include the complete attributes of its associated resource. Compared to distributed inverted index, Midas assigns only one key to a d-attribute resource such that query processing does not need to create intermediate result sets.

d-to-d mapping scheme such as pSearch [13], MURK [14], and 2CAN [15], maps a d-attribute resource to a coordinate point in a d-dimensional space. Thus, a multi-attribute range query corresponds to a region in the multi-dimensional space. A search request is first routed to any point in the query region, and then propagated to the remaining points in the region. d-to-one mapping requires an underlying DHT that supports multidimensional space, e.g. CAN [21]. However, Midas can be applied on one-dimensional DHT.

In d-to-one mapping scheme such as Squid [16], CONE [15], ZNet [17], SCRAP [14] and CISS [18], we consider a d-attribute resource as a coordinate point in a d-dimensional attribute space. The point is mapped to a key in a one-dimensional identifier space using a d-to-one mapping function such as space-filling curve (SFC). To process a query, we apply a d-to-one mapping function to the query to obtain a number of search keys. To reduce the cost per query in terms of number of nodes visited, query processing exploits the property of conventional DHT where each node is responsible for several unique keys. Though Midas is also based on d-to-one, each R-DHT node is responsible only for one unique key. Hence, Midas optimizes query processing by performing lookups only for available keys.

III. Midas Design

Midas is an approach to support multi-attribute range queries on R-DHT, based on d-to-one mapping scheme. Midas consists of two main parts (Figure 1): an *indexing* scheme maps a multi-attribute resource to an R-DHT node, and a *query processing* scheme to lookup available keys. Midas uses Hilbert space-filling curve [22] as the d-to-one mapping function.

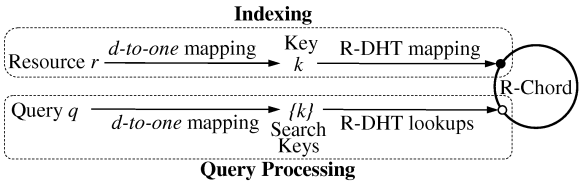


Fig. 1. Midas Indexing and Query Processing

In the following, we present an overview of R-Chord and the design of Midas.

A. Overview of R-Chord

To achieve the read-only property where a key-value pair is stored on its originating location, R-Chord virtualizes each *host*[3] into *nodes*[4] by associating node $n_{k,h}$ to each unique key k owned by host h. Thus, R-Chord virtualizes h into $|T_h|$ nodes where T_h denotes the set of unique keys owned by h.

R-Chord uses two different identifier spaces: an m-bit identifier space for both keys and host identifiers[5], and a $2m$-bit identifier space for node identifiers. A $2m$-bit node identifier, denoted as $k|h$, is formed by concatenating key k and host identifier h (Figure 2). The $2m$-bit identifier spaces prevents collisions of node identifiers when several hosts share the same keys.

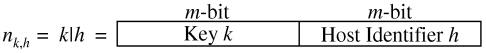


Fig. 2. $2m$-bit Node Identifier Space

R-Chord organizes nodes as a Chord ring and divides the ring into segments based on the keys (i.e. resource types). With segment addressing, R-Chord supports the flat-naming scheme that abstract the location of keys in the lookup interface, i.e. *lookup(key)*. In addition, segment-based organization increases lookup resiliency whereby key k can still be found even if some nodes in segment S_k fail. This is because S_k consists of nodes with the same prefix k (i.e. the same resource type) but a different suffix (i.e. belonging to different hosts).

Figure 3 illustrates the mapping scheme of R-Chord. In this example, host A owns two unique keys, i.e. $T_A = \{2, 9\}$, and thus, we virtualize host A into two nodes: node $2|A$ and node $9|A$. Similarly, we virtualize host B and host C into one node and three nodes, respectively. The ring overlay of R-Chord consists of three segments, namely segment S_2 consisting of node $2|A$ and node $2|C$, segment S_5 consisting of node $5|B$ and $5|C$, and segment S_9 consisting of node $9|A$ and node $9|C$. These segments represents the three keys: 2, 5, and 9.

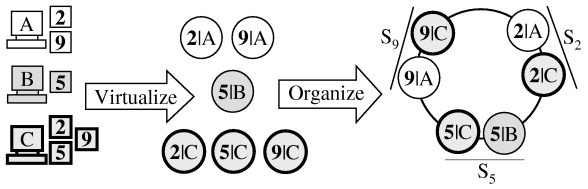


Fig. 3. R-Chord Virtualizes Three Hosts into Six Nodes

In a system with N hosts and K unique keys[6], the lookup path length in R-Chord is $O(\min(\log K, \log N))$ hops [2]. The lookup algorithm is based on Chord lookup in which (i) every node n maintains a *finger table* consisting of $O(2m)$ entries, and (ii) the i^{th} entry (*finger*) points to the successor of $(n + 2^{i-1})$. However, we improve the basic Chord lookup by exploiting R-Chord key-to-node mapping scheme:

1) **Routing by segments**: To locate key k, we route a lookup request from one segment to another. Each routing step halves the distance, in terms of *segments*, to the destination segment S_k. Thus, the lookup path length is $O(\log K)$.

[3]A physical entity that shares key-value pairs.
[4]A logical entity in the context of overlay network.
[5]For example, hash of h's IP address.
[6]$K = |\bigcup_{h=0}^{N-1} T_h|$

2) **Shared finger tables**: To limit the lookup path length at $O(\log N)$ hops even when $K > N$, each routing step utilizes all the $|T_h|$ finger tables maintained by host h. In other words, a node's finger table is shared by all nodes belonging to the same host. Thus, visiting one host is equivalent to visiting all the nodes corresponding to the host. Since the maximum distance between two segments is $O(N)$ hosts, it takes $O(\log N)$ hops to locate a segment.

To improve lookup resiliency due to node failures, R-Chord nodes maintain *backup fingers* so that when a finger fails, i.e. points to a non-existing node, we promote a backup finger to replace failed fingers (see Figure 4). A possible implementation of backup fingers is to piggyback periodic finger corrections (see [19] for detail). When n corrects its finger f, in addition to $successor(f)$, the correction process also returns the fingers that share the same prefix as f.

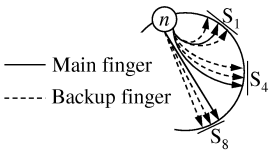


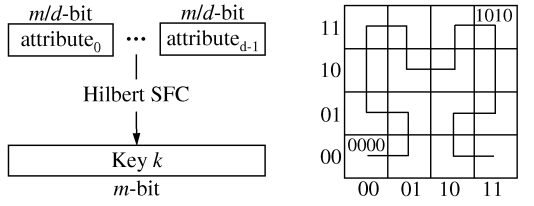
Fig. 4. Backup Fingers

B. Multi-Attribute Indexing

Using Hilbert space-filling curve (SFC), Midas maps a d-dimensional attribute space to a one-dimensional identifier space. Figure 5 shows a resource with d attributes being mapped onto an m-bit key[7] where each attribute is represented as an (m/d)-bit value (Figure 5(a)). The resource is modeled as a coordinate point in a d-dimensional attribute space. Each dimension represents an attribute type, and each (m/d)-bit dimension value represents an attribute value. A Hilbert SFC maps the coordinate point to an m-bit key in a one-dimensional identifier space. Figure 5(b) illustrates the mapping of $(00_2, 00_2)$ and $(11_2, 11_2)$ to 0000_2 and 1010_2, respectively. One criteria to evaluate the performance of SFC is the *clustering* property: two one-dimensional points that are close in the one-dimensional space represents two d-dimensional points that are close in the d-dimensional space. Jagadish [23] has shown that for multi-dimensional indexing and range query, Hilbert SFC minimizes the number of clusters, compared to other types of SFC.

Figure 6 illustrates an example of Midas indexing. Assume that resources are described by two attributes, `cpu` and `memory`. Using a 4-bit key, each attribute occupies 2-bit in a two-dimensional attribute space. Resource r with attributes `cpu` = *P3* and `memory` = *1 GB* is represented as coordinate (3, 0) in the two-dimensional space, where 3 in `cpu`-dimension and 0 in `memory`-dimension represents `cpu`-value *P3* and `memory`-value *1 GB*, respectively. This coordinate is translated using Hilbert SFC to a one-dimensional key with a value

[7]Each unique key represents a *resource type*.

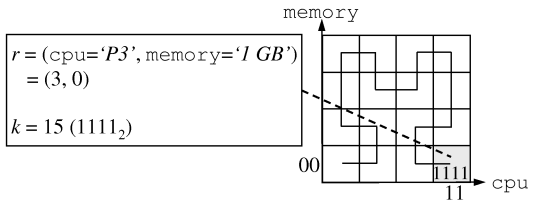


(a) Attributes and Key

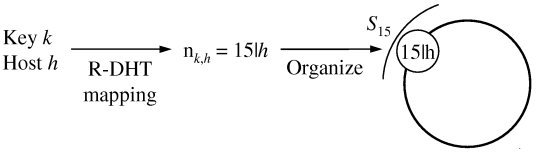
(b) Hilbert SFC on Two-Dimensional Space (m = 4-bit)

Fig. 5. Midas Indexing using Hilbert SFC

of 15 (1111_2), and Midas assigns this key to r (Figure 6(a)). Lastly, through the R-DHT mapping scheme, Midas associates node $n_{15,h}$ to r, where h is the host sharing resource r, and the node occupies segment S_{15} in an R-Chord overlay (Figure 6(b)).



(a) Assign Key 15 to Resource r

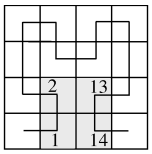


(b) Map Key 15 to R-DHT Node

Fig. 6. Example of Midas Indexing (m = 4-bit)

C. Query Processing and Optimization

Midas transforms a d-attribute range query into search keys. Each range query corresponds to a region in a d-dimensional attribute space (Figure 7). Using Hilbert SFC, this query region is tranformed into a number of one-dimensional *search keys* where consecutive search keys form a cluster. Then, Midas locates the nodes responsible for the search keys using one or more DHT lookups.



Four search keys {1, 2, 13, 14} grouped in two clusters, i.e., cluster 1–2 and 13–14.

Fig. 7. Transforming a Range Query into Search Keys

In a naive search, a query initiator (i.e. the user) issues one lookup per search key. However, this is not efficient:

1) The naive search includes unnecessary lookups for search keys that do not represents resources. The example in Figure 8 shows two unnecessary lookups for

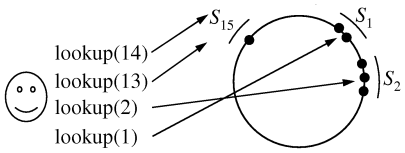


Fig. 8. Naive Search Algorithm

search keys 13 and 14, out of four lookups. Because search keys 13 and 14 do not correspond to resources, there are no S_{13} and S_{14} in the underlying R-Chord overlay. As a result, both the lookups terminate at S_{15}.

2) The naive search ignores the clustering property of Hilbert SFC. Since search key 1 and 2 are clustered, we can exploit the close proximity between S_1 and S_2 by initiating *lookup(1)* only and let S_1 continues with another lookup request to S_2.

To support efficient query processing, we propose an incremental search that processes only *available keys*. The *mdq_search()* algorithm[8] is shown in Figure 9. The algorithm transforms a query region[9] into a sorted list of search keys. The incremental search then starts by initiating a *lookup(k)* for the lowest key k in the list. This lookup will end-up in a segment $S_{k'}$ that succeeds k. If $k' = k$ then we add k (or the key-value pair) to the result set, otherwise we discard k. Before continuing, we remove search key k" that satisfies one of the following conditions:

1) $preceeding\_segment(k) < k" < k'$
   This eliminates unavailable key that precedes k', because $S_{k'}$ is the succeeding segment of k" only if k" does not exist. We modify *lookup(k)* so that it returns the succeeding segment and the preceding segment of k.
2) $k' < k" < succeeding\_segment(k')$
   This eliminates unavailable key k" that succeeds k'. To quickly locate $S_{k'}$, each node in the overlay maintains a pointer to its succeeding segment. Thus, it is suffice to examine the node's finger table instead of issuing a *lookup*$((S_k + 1)|0)$ request.

Figure 10 shows an example of incremental search for the query illustrated in Figure 7. Given the four search keys, 1, 2, 13, and 14, Midas initiates a DHT lookup for the lowest key 1. Since the lookup finds the key at segment S_1, Midas adds key 1 to the result set and continues with *lookup(2)*. As this lookup arrives at S_2, Midas will add key 2 to the result set. Furthermore, it eliminates key 13 and 14 since the succeeding segment of S_2 is S_{15}. Thus, the final result set consists of key 1 and key 2. To return query results faster to query initiators, the query processing can be parallelized by partitioning the search keys and performing one incremental search per partition.

[8]Remote procedure calls or variables are preceeded by the remote node identifier, while local procedure calls and variables omit the local node identifier.

[9]A query region is described by its smallest and largest coordinates, i.e. the two endpoints of a diagonal. The smallest/largest coordinate is the coordinate where each dimension consists of the smallest/largest value (of the respective dimension) in the query region. As an example, the smallest and largest coordinates for the query example in Figure 7 are (1, 0) and (2, 1), respectively.

```
h.mdq_search(Point smallest, Point largest)
    q = {} ∪ HilbertSFC(smallest, largest);
    rs = {};
    k = get_min(q);
    (n, p) = lookup(k);
    return n.incr_search(q, rs, p);
n.incr_search(List q, ResultSet rs, PreceedingSegment p)
    //Check if h owns a key equals to search key
    k = get_min(q);
    T_h = a set of keys in h;
    for each y ∈ T_h do
        if y == k then
            rs = rs ∪ {(k, n)};

    q = q − {k};
    q = eliminate_keys(q, p, prefix(n));
    q = eliminate_keys(q, prefix(n), prefix(succ_seg));
    if q == {} then
        return rs;

    //Search the next lowest key
    k = get_min(q);
    (n', p) = lookup(k);
    return n'.incr_search(q, rs, p);
n.eliminate_keys(List q, Key low, Key high)
    k = get_min(q);
    while k ≠ nil and low < k < high do
        q = q − {k};
        k = get_min(q);

    return q;
```

Fig. 9. Midas Search Algorithm

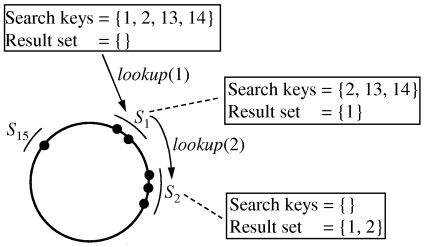


Fig. 10. Example of Range Query Processing

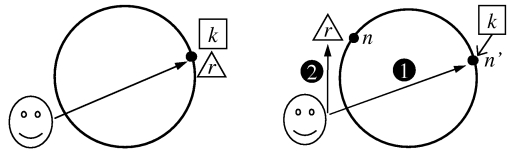
R-Chord allows a user to access resources without sending an additional request, because resource r and its key k are located at the same node (Figure 11(a)). In contrast, on conventional Chord, key k belonging to node n is mapped onto another node n'. To access resource r, a user first locates k at n' before accessing r at node n (Figure 11(b)).

IV. Evaluation

To study the performance of multi-attribute range queries on DHT, we implement Midas using R-Chord and conventional Chord[10] as the underlying overlay topology. Using simulation, we evaluate three metrics: lookup resiliency, query cost, and lookup performance under churn. Unless stated otherwise, our experiments use the following settings:

- m, the number of bits for keys and host identifiers, is 18-bit.

[10]In this implementation, Midas uses only the first condition in eliminating search keys (Section III-C), where $preceeding\_segment$ is replaced by $preceeding\_node$.



(a) Resource r and Key k are Located at the Same Node (R-Chord)

(b) Resource r and Key k are Separated (Chord)

Fig. 11. Locating Key and Accessing Resource in R-Chord and Chord

- d, the number of attributes per resource, is varied from 3 to 4.
- K, the number of unique keys (resource types), is varied from 1,000 to 50,000. This means, there are K points in the d-dimensional attribute space. We generate these points using the normal distribution.
- N, the number of hosts, is 25,000. Each host shares $|T_h| \sim U[4, 10]$ unique resource types, and each resource type may be offered for sharing by more than one hosts.
- Each range query covers 1% of the d-dimensional attribute space. This means, the number of search keys per query is $0.01 * 2^m$. Such *query selectivity* has also been used in other works [14], [16].

A. Resiliency to Node Failures

Lookup is resilient when it is able to locate keys in the presence of node failures. To evaluate the lookup resiliency, we simulate 200,000 range queries for a different percentage of node failures (F). Figure I shows the percentage of available keys successfully retrieved in R-Chord and Chord.

TABLE I
LOOKUP RESILIENCY UNDER NODE FAILURES

F	K	% Keys Retrieved			
		d = 3		d = 4	
		Chord	R-Chord	Chord	R-Chord
20%	1,000	73	100	76	100
	10,000	75	100	76	100
	50,000	78	99	77	100
40%	1,000	45	99	47	99
	10,000	47	99	46	99
	50,000	51	97	47	98

Our results show that lookup resiliency is higher in R-Chord; nearly all available keys are retrieved. We attribute this to the read-only property of R-Chord. Because each R-Chord node is responsible only for its own keys, when a node fails, only its own keys are affected. And by its design (i.e. routing by segments and backup fingers), R-Chord can locate a key as long as there is at least one alive node still sharing the key. On the other hand, Chord stores a key belonging to one node on another responsible node. When the responsible node fails, Chord fails to locate the keys stored on the responsible node, even is the originating node of the keys is still alive.

B. Cost of Query Processing

The cost of query processing depends on two factors: number of search keys per query, and number of available keys per query. In our experiments, the number of search keys per query is constant while the number of available keys per query is affected by K (Table II).

TABLE II
AVERAGE NUMBER OF AVAILABLE KEYS PER QUERY

F	K	# Available Keys	
		d = 3	d = 4
20%	1,000	5	44
	10,000	49	346
	50,000	220	716
40%	1,000	5	44
	10,000	49	346
	50,000	206	668

Table III shows that the average number of nodes visited per query in R-Chord is affected by K. However, Midas visited a constant number of Chord nodes per query. We attribute this to the impact of available keys and search keys.

TABLE III
AVERAGE NUMBER OF NODES VISITED PER QUERY

F	K	# Nodes Visited			
		d = 3		d = 4	
		Chord	R-Chord	Chord	R-Chord
20%	1,000	154	63	202	221
	10,000	154	160	206	592
	50,000	154	357	207	945
40%	1,000	113	60	153	213
	10,000	113	154	149	580
	50,000	112	336	145	908

Because each R-Chord node is responsible for its own key, the number of nodes visited is at least equal to the number of available keys. Hence, when the available keys increases due to a larger K, so does the number of nodes visited. In Chord, the number of nodes visited is constant, irrespective of K, due to the followings:

1) Chord visits nodes responsible for search keys (Figure 12).
   The number of such nodes is affected only by the number of search keys per range query and N.

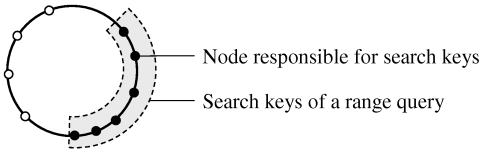


Fig. 12. Chord Nodes Responsible for Search Keys

2) The query cost excludes additional hops needed to access resources.
   The number of additional hops is at most equal to the number of available keys shown in Table II. Thus, the total cost becomes the sum of query cost and the additional hops to access resources. For example, when $K = 50,000$ and $d = 4$, the total cost is $207+716 = 923$ hops.

C. Performance of Range Queries under Churn

The virtualization of a host into nodes increases the size of R-Chord overlay. In addition, it magnifies the impact of churn because each host arrival, fail, and leave will result in simultaneous node joins, fails, and leaves[11], respectively.

To evaluate range query processing under churn, we compare Midas on R-Chord and Chord overlays that change dynamically. We begin the experiments by warming up 25,000 hosts. In the next one-hour period, we simulate arrivals of 25,000 new hosts, in addition to fails and leaves of 25,000 hosts. Thus, there will be $N \sim 25,000$ alive hosts at any time within this duration. During this one-hour period, we also simulate a number of range query events, where the ratio of arrive:fail:leave:query is set at 2:1:1:6. Assuming that these events follow a Poisson distribution, we derive a rate of 10 events/second based on the measurements on peer life-time by Bhagwan et. al. [24]. Each node in the overlay invokes the finger correction every 60 seconds on average. Table IV presents the percentage of keys successfully retrieved.

TABLE IV
PERFORMANCE OF RANGE QUERIES UNDER CHURN

K	% Keys Retrieved			
	d = 3		d = 4	
	Chord	R-Chord	Chord	R-Chord
1,000	99	99	98	98
10,000	97	98	94	97
50,000	95	89	94	88

The result shows that R-Chord performs reasonably well under churn. Though R-Chord overlay is seven times larger than Chord, the number of available key retrieved in R-Chord is only 6% lower than Chord when $K = 50,000$. The reason of R-Chord retrieving less keys when $K = 50,000$ is because each segment in the overlay consists of only a small number of nodes (about 3.5 nodes per segment). This reduces the effectiveness of backup fingers.

V. CONCLUSION

We have presented Midas, a scheme to support multi-attribute range queries on R-DHT. Using Hilbert SFC, Midas assigns to each d-attribute resource a one-dimensional m-bit key. In processing a range query, Midas performs search-keys elimination to avoid issuing unnecessary lookups, and incremental search to exploit the clustering property of Hilbert SFC. Due to its read-only property, R-DHT does not need to send additional requests to access resources, as resources and its key are co-located.

Our simulation evaluation shows that R-DHT increases lookup resiliency; nearly all available keys are found even when 40% of nodes simultaneously fail. The cost of query processing in R-DHT is affected by the number of available keys rather than the query selectivity. This indicates that R-DHT is suitable in applications where query selectivity is much larger than the number of query answers.

[11]When a node leaves, it notifies its successor and predecessor to adjust their fingers accordingly. In addition, a node leaving a Chord ring migrates all keys stored onto it to its successor; this migration completes without delay.

REFERENCES

[1] B. T. Loo, R. Huebsch, I. Stoica, and J. M. Hellerstein, "The case for a hybrid P2P search infrastructure," in *Proc. of the 3rd IPTPS*, Feb. 2004, pp. 141–150.

[2] Y. M. Teo, V. March, and X. Wang, "A DHT-based grid resource indexing and discovery scheme," in *Proc. of Singapore-MIT Alliance Annual Symposium*, Jan. 2005.

[3] N. Daswani, H. Garcia-Molina, and B. Yang, "Open problems in data-sharing peer-to-peer systems," in *Proc. of the 9th ICDT*, Jan. 2003, pp. 1–15.

[4] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, "SkipNet: A scalable overlay network with practical locality properties," in *Proc. of the 4th USITS*, Mar. 2003, pp. 113–126.

[5] A. Mislove and P. Druschel, "Providing administrative control and autonomy in structured peer-to-peer overlays," in *Proc. of the 3rd IPTPS*, Feb. 2004, pp. 162–172.

[6] B. F. Cooper, "Quickly routing searches without having to move content," in *Proc. of the 4th IPTPS*, Feb. 2005, pp. 163–172.

[7] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish, "A layered naming architecture for the internet," in *Proc. of ACM SIGCOMM*, Sept. 2004, pp. 343–352.

[8] M. Cai, M. Frank, J. Chen, and P. Szekely, "MAAN: A Multi-Attribute Addressable Network for grid information services," *Journal of Grid Computing*, vol. 2, no. 1, pp. 3–14, 2004.

[9] D. Bauer, P. Hurley, R. Pletka, and M. Waldvogel, "Bringing efficient advanced queries to distributed hash tables," in *Proc. of the 29th LCN*, Nov. 2004, pp. 6–14.

[10] M. Harren, J. M. Hellerstein, R. Huebsch, B.T.Loo, S. Shenker, and I. Stoica, "Complex queries in DHT-based peer-to-peer networks," in *Proc. of the 1st IPTPS*, Mar. 2002, pp. 242–249.

[11] O. D. Gnawali, "A keyword-set search system for peer-to-peer networks," Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, May 2002.

[12] S. Shi, G. Yang, D. Wang, J. Yu, S. Qu, and M. Chen, "Making peer-to-peer keyword searching feasible using multi-level partitioning," in *Proc. of the 3rd IPTPS*, Feb. 2004, pp. 151–161.

[13] C. Tang, Z. Xu, and S. Dwarkadas, "Peer-to-peer information retrieval using self-organizing semantic overlay networks," in *Proc. of ACM SIGCOMM*, Aug. 2003, pp. 175–186.

[14] P. Ganesan, B. Yang, and H. Garcia-Molina, "One torus to rule them all: Multi-dimensional queries in P2P systems," in *Proc. of the 7th WebDB*, June 2004, pp. 19–24.

[15] D. Agrawal, A. E. Abbadi, and S. Suri, "Attribute-based access to distributed data over P2P networks," in *Proc. of the 4th DNIS*, Mar. 2005, pp. 244–263.

[16] C. Schmidt and M. Parashar, "Flexible information discovery in decentralized distributed systems," in *Proc. of the 12th HPDC*, June 2003, pp. 226–235.

[17] Y. Shu, B. C. Ooi, K.-L. Tan, and A. Zhou, "Supporting multi-dimensional range queries in peer-to-peer systems," in *Proc. of the 5th P2P*, Aug. 2005, pp. 173–180.

[18] J. Lee, H. Lee, S. Kang, S. Choe, and J. Song, "CISS: An efficient object clustering framework for DHT-based peer-to-peer applications," in *Proc. of DBISP2P*, Aug. 2004, pp. 215–229.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. of ACM SIGCOMM*, Aug. 2001, pp. 149–160.

[20] A. Andrzejak and Z. Xu, "Scalable, efficient range queries for grid information services," in *Proc. of the 2nd P2P*, Sept. 2002, pp. 33–40.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable Content-Addressable Network," in *Proc. of ACM SIGCOMM*, Aug. 2001, pp. 161–172.

[22] H. Sagan, *Space-Filling Curves*. Springer-Verlag, 1999.

[23] H. V. Jagadish, "Linear clustering of objects with multiple attributes," in *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1990, pp. 332–342.

[24] R. Bhagwan, S. Savage, and G. M. Voelker. "Understanding availability," in *Proc. of the 2nd IPTPS*, Feb. 2003, pp. 256–267.