The RangeGuard: Range query optimization in peer-to-peer data networks

Nikos Ntarmos, Theoni Pitoura, and Peter Triantafillou

2005
The RangeGuard: Range query optimization in peer-to-peer data networks

Nikos Ntarmos Theoni Pitoura Peter Triantafillou
Research Academic Computer Technology Institute and
Computer Engineering & Informatics Dept., University of Patras, Greece
{ntarmos,pitoura,peter}@ceid.upatras.gr

Abstract

In this work we address the issue of efficient processing of range queries in DHT-based peer-to-peer data networks. The novelty of the proposed approach lies on architectures, algorithms, and mechanisms for identifying and appropriately exploiting powerful nodes in p2p data networks, the existence of which has been well documented in the literature. We first present a framework that consists of a modified DHT architecture, data placement strategies, and search algorithms, providing efficient support for processing range queries. Subsequently, we extend this framework with mechanisms to identify powerful nodes, and seamlessly and efficiently incorporate them within this framework, so to further improve range query processing. The significant performance improvements are due to (i) ensuring a much smaller hop count performance for range queries, and (ii) avoiding the dangers and inefficiencies of relying for range query processing on weak nodes, with respect to processing, storage, and communication capacities, and with intermittent connectivity. We also present experimental results validating our performance claims.

1 Introduction

Structured P2P systems have provided the P2P community with efficient and combined routing/location primitives. This goal is accomplished by maintaining a structure in the system, emerging by the way that peers define their neighbors. These systems are usually built around the notion of a distributed hash table (DHT). Prominent examples of such architectures include Chord [21], CAN [16], Pastry [18], Kademlia [12], SkipNet [7], etc.

DHTs have managed to take routing and location of data items in P2P systems to the next level; from the undetermined, flood-based techniques used in early, unstructured P2P overlays, DHTs provide us with strong probabilistic guarantees on the worst-case number of hops required to route a message from a node to any other node in the system or, equivalently, for a node in the system to locate data items published therein. Unfortunately, all currently available DHT overlays are designed to only support single-identity, exact-match queries. This has lead researchers to investigate how they could enhance P2P systems to reply to more complex queries; however, up to now support for range query optimization remains an open problem.

On the other hand, one of the main characteristics of widely deployed P2P networks (e.g. Gnutella [3], Mojonation [13], etc.) is that participating peers are largely heterogeneous, with regard to their processing power, available main memory and disk storage, network bandwidth, and internet connection uptime. Relevant studies of P2P networks [1,20] have shown that this large heterogeneity is also depicted in the distribution of the query processing chores across the node population. Namely, in the Gnutella [3] network, about 70% of the nodes share no files at all with the community, while 5% of the nodes in the system serve almost 95% of the queries posed.

Therefore, we believe that harnessing the power of such powerful and “altruistic” nodes is the key to providing efficient range query processing in a P2P setting. With this work we will also present solutions for efficiently supporting range queries, by leveraging the extra capabilities of the powerful nodes existing in a P2P overlay. To our knowledge, this is the first work to look into this issue. Given the prominence of DHT-based architectures, we will assume a popular DHT (Chord) as our substrate, hoping that in this way our work can leverage existing results.

2 An Overview of Chord

Chord uses an \(m\)-bit circular identifier space for nodes and documents and modulo-\(N\) arithmetic (with \(N = 2^m\) being the maximum number of nodes/objects in the system). In order to place documents on nodes, Chord makes use of consistent hashing [10]; a document with ID \(d\) is placed on the node whose ID is the least one that is greater or equal to \(d\) on the identifier circle. This node is called the successor of \(d\), denoted \(\text{succ}(d)\). Nodes on a Chord ring maintain
log(N) links to other nodes, in what is called the node’s finger table. The kth entry in the finger table (also called the order-k successor) of a node with ID n, points to the successor of n + 2k−1. Each node also maintains a link to its immediate predecessor. Furthermore, for fault-tolerance reasons, each node maintains log(N) order-1 (i.e. immediate) successors, for a total of O(log(N)) links per node.

When a Chord node wishes to route a message to a node with ID n, it uses its highest-order finger entry corresponding to a node whose ID precedes n. This allows nodes in a Chord ring to route messages to any other node in O(log(N)) hops in the worst case. However, due to the dynamics of the peer population, any successor of a node may fail or leave the system at will. Chord is based on the assumption that not all log(N) order-1 successors of a node will ever fail or leave the system simultaneously (i.e. the resulting graph is log(N)-robust). When the node population is highly dynamic (i.e. finger table entries are wrong for most nodes in the system), Chord nodes may still route requests, using their order-1 successor pointers, for an O(N) worst-case hop-count. Thus, in order for Chord to function efficiently, finger table entries must be as up-to-date as possible. This is accomplished through a periodic process, called stabilization in Chord’s terminology; nodes probe their successors, check whether the corresponding order-k successor is indeed the successor of their ID plus 2k−1, and take proper action so that the latter always holds.

In order to achieve a balanced load on the participating nodes, Chord distributes documents to nodes in a rather aggressive manner; both node and document IDs are based on cryptographic hashing (SHA-1) of a node/document specific piece of information. This results in a quasi-uniform distribution of nodes and documents on the identifier circle, carrying on to the distribution of documents to nodes, since documents are assigned to nodes using consistent hashing.

Now assume that every document has a distinct “index” value associated with it (a la tuple indices in relational databases), and that this value is used as the input to SHA-1. This would result in documents being spread across all participating nodes in a uniform manner, but would also lead to documents with successive index values being stored on completely unrelated nodes. This fact renders Chord highly inefficient for range query processing; given a range on the domain of the index values, Chord has to execute queries for each and every value in the range interval! A range query for r consecutive values would require on average r queries to be executed, for a total of O(rlog(N)) hops.

In a non-densely populated value domain, most of these queries would return no data items. Assume now that all nodes are aware, in some obscure “magic” way, of every value stored in the P2P overlay (we’ll call this system the Enhanced Chord). They could then skip queries for values not existing on any node in the system. Then, given the above range query, if only r′ out of r values are stored in the system, it would take on average r′ queries, for a total of O(r′ log(N)) hops. Although this is better than O(r log(N)), it still is too expensive for a real-world system. In the next sections we introduce algorithms and protocols extending Chord so that such queries will require at most O(log(N)+r′) hops, and describe a multi-level architecture that further improves this hop-count by orders of magnitude.

3 OP-Chord: Speeding Up Range Queries in the Chord Overlay

We assume that data stored in the P2P network is structured in a (k + l)-attribute relation R(a1, . . . , ak, b1, . . . , bl), where ai, bi are the attributes of R, with every tuple t in R being uniquely identified by a primary key t.key. This key can be either one of the attributes of the tuple, or can be calculated otherwise (e.g. based on the values of one or more of the attributes of t). Furthermore, attributes ai are used as single-attribute indices of t, with each ai being characterized by the domain t.ai.D : {t.ai,vmin,t.ai,vstep,t.ai,vmax} of its values. In the proposed framework we will use Chord, because of its simplicity, its popularity within the P2P community, but also because of its good performance (logarithmic in terms of routing hops and routing state), robustness, and fault-tolerance.

3.1 Data Insertion

A tuple t with key t.key is stored in the network using Chord’s consistent hashing, in peer succ(t.key). In addition, for every index attribute t.ai of t, with value t.ai,v, we store an index tuple I(t.ai) : {t.ai,v,t.ai.id,t.key}, where t.ai.id is the attributes unique identifier. Optionally, instead of t.key the tuple itself (replica) may be stored. Therefore, for each tuple inserted in the network, we also store k index tuples. These index tuples are stored at nodes succ(hi(t.ai,v)), using the following mod 2m order preserving hash function (OPHF):

\[ h_i(t.ai,v) = (t.ai,s_0 + t.ai,v - t.ai,v_{\text{min}} \times 2^m) \mod 2^m \]

where t.ai,s0 = hash(t.ai,id) is used to randomize the point on the identifier circle to which t.ai,vmin will be mapped (to compensate for multiple identical attribute domains or for multiple domains starting from the same value, and hash() is the base hash function used by the underlying DHT (e.g. SHA-1). If t.ai.D is the attribute value domain and N is the node ID domain, then hi() is a one-to-one mapping from the former to the latter – hi() : t.ai.D → N. Deletions and updates of the original tuples are broadcast to
the peers holding the index tuples. Note that since the quantities \( t \cdot a_i \cdot v\{min, max\} \) and \( t \cdot a_i \cdot s_0 \) are different for each attribute \( a_i \), there is a different \( h_i() \) for every attribute. We refer to the above as the OP-Chord architecture, as it is largely defined by the order-preserving hash function used.

An alternative to storing index tuples, would be to have inserted tuples fully replicated to the positions dictated by the above algorithm. To maintain \( k \) replicas of each tuple may be inefficient, as far as storage requirements are concerned and requires more processing time for data updates to keep replicas mutually consistent, as well as more overhead when nodes join/leave the network. On the other hand, it is indisputable that replicas of data in a DHT network are required given the frequent topology changes, if data availability is to remain at reasonable levels. Furthermore, this approach keeps routing hops for lookups significantly low, since range queries can be processed more efficiently as we will describe below. For the rest of this paper we assume that index tuples contain replicas of the indexed tuples.

**Example 1 (Data placement in OP-Chord)** Assume we have a 12-bit identifier space (i.e. node IDs are in the interval \([0, 4095]\)), and tuples with one index attribute receiving integer values in the interval \([1, 999]\) (with \( s_0 = 0 \) for simplicity reasons). Now assume that our P2P overlay consists of 9 nodes with IDs: 0, 340, 690, 1024, 1540, 2048, 2560, 3072, and 3590. Our OPHF would then be defined by the order-preserving hash function used.

Given that the ring depicted is a Chord ring (thus using consistent hashing to map values to nodes), every node in the system is responsible for the value interval immediately preceding it (e.g. node 1024 is responsible for the value interval \([169, 250]\), node 2560 is responsible for the value interval \([501, 625]\), etc.) Thus, an index tuple with a value of 34 would be stored on node 340, while an index tuple with a value of 667 would be stored on node 3072, etc.

### 3.2 Range Query Processing

Given a range query \((a_i, v_{low}, a_i, v_{high})\), with \( a_i, v_{high} = a_i, v_{low} + r \), for an attribute \( a_i \), we proceed as follows. We first apply our OPHF on \( v_{low} \) and send the query to peer \( p_1 : \text{succ}( h_i( a_i, v_{low} ) ) \). By design (because of the order-preserving hashing), the requested index tuples are stored in peers falling between \( p_l : \text{succ}( h_i( a_i, v_{low} ) ) \) and \( p_h : \text{succ}( h_i( a_i, v_{high} ) ) \) inclusive. Thus, every peer \( p_j \) in this arc of the Chord ring \((l \leq j \leq h)\): (i) first sends back to the requester any locally present index tuples that satisfy the query, and (ii) forwards the query to its best-fitted successor node. The latter is found as follows. If we assume that the Chord ring is in a steady state, then we can use the inverse function \( h_i^{-1}() : N \rightarrow a_i, D \) of the OPHF:

\[
h_i^{-1}(n_i) = a_i, v_{min} + \frac{(n_i - a_i, s_0) \mod 2^m}{2^m} \times (a_i, v_{max} - a_i, v_{min})
\]

to compute the highest value of \( a_i \) for which any node \( n_i \) is responsible, and forward the query to the node \( \text{succ}( h_i^{-1}(n_i) ) \) responsible for the value immediately succeeding \( h_i^{-1}(n_i) \).

**Example 2 (Range query processing in OP-Chord)**

Assume we have the same Chord ring as in ex. 1 above, and that we want to execute a range query for the interval \([400, 650]\). Using our OPHF we find that \( v_{low} = 400 \) is stored on node with ID \( p_l = 2048 \) (responsible for the interval \([376, 500]\)), and send the query to it (fig. 1).

Node 2048 returns any locally stored index tuples to the requesting node, and forwards the query to its successor. Forwarding stops when we reach the node with ID \( p_h = 3072 \), responsible (according to OPHF) for \( v_{high} = 650 \) (lying in its interval of responsibility – \([626, 750]\)).

### 3.3 Expected Performance Benefits

Since \( r \) values fall in the requested range, at most \( r \) successor pointers will be followed, bringing the total hop count to \( O(\log(N) + r) \); \( O(\log(N)) \) hops to reach the node \( p_l \), plus \( r \) hops to reach all other nodes. This compares very well to \( O(r \log(N)) \) that would be required to access each individual node in any basic DHT architecture. In sec. 4, we enhance our architecture using a number of special, powerful peers: the RangeGuards.

### 4 The RangeGuard

We form a second OP-Chord ring, the RangeGuard ring, composed of powerful nodes burdened with extra routing state and functionality – the RangeGuards – above the OP-Chord ring containing all nodes of the network (fig. 2).
Each such node is responsible for storing the index tuples placed in nodes between its predecessor RangeGuard and itself. Thus, if there are $M$ RangeGuards (RGs) in the system, they partition, using consistent hashing, the normal Chord ring into $M$ continuous and disjoint arcs (ranges).

Each RG maintains routing information for both the lower-level ring and the RangeGuard ring. Additionally, there is a direct link from each peer to the next RG in the upper-level ring (i.e. to the RG responsible for the peer). In this setting, nodes in the lower-level ring probe their RG (e.g. as part of the standard Chord stabilization process), and automatically update the index tuples it stores.

Much like DHTs, we assume that peers won’t act maliciously or selfishly (leaving countermeasures for such behavior as a possibility for future work, as is the case for most DHT-based research). However, RangeGuards must be (i) powerful enough and willing to withstand the extra (storage, processing, communication) load, and (ii) connected most of the time, to provide hop-count guarantees for range queries and to avoid large transfers due to their joining/leaving. This, in turn, calls for a mechanism to identify and promote candidate RangeGuards in an efficient and transparent manner. Given the functionality offered by our initial (without RangeGuards) infrastructure, RangeGuards are identified and located as follows.

### 4.1 Node Performance Counters and the Node Performance Relation (NPR)

The administrator of each node selects whether she wants her node to be a candidate for RangeGuard membership or not. Then, each candidate node $n$, with id $n.id$, keeps track of the amount $n.\alpha$ of retrieval and/or just routing requests it serves. This information is updated periodically, every $\mathcal{E}$ seconds (also called an *epoch*). Thus, each node keeps two node performance counters (or NPCs) – $n.\alpha_c$ and $n.\alpha_p$ – of requests served during the current and the previous epoch respectively.

This information is stored in the system as a three-attribute node performance relation $\mathcal{NPR} : \{n.id, n.\alpha_p, \mathcal{U}, status\}$, with primary key $n.id$, and indexed by $n.\alpha_p$. $status$ is a boolean variable, set to true if the node is a part of the RangeGuard. $\mathcal{U}$ is a counter, incremented on every update of the tuple, and left-shifted (assuming a little-endian architecture) (i) on every update or (ii) every $\delta + \mathcal{E}$ seconds, with the timer being reset on every update. $\delta$ is a quantity depending on measurable characteristics (e.g. round-trip / ping time) of the end-to-end connection between node $n$ and the node storing the tuple with $n$.metadata. $\mathcal{U}$ encapsulates the amount of time a peer stays connected to the network.

Note that if a node wishes to cease being a candidate for RangeGuard, it suffices to set a low value (e.g. 0) for its $\alpha$ and stop updating this information (so that its $\mathcal{U}$ value decays to 0 after some time of inactivity). Also note that the $\mathcal{NPR}$ tuple of a node $n$ is stored on this very node, since the primary key of the relation is the $n.id$ attribute.

#### Cost of Maintaining NPR

Candidate RG nodes must update the index tuple for their $\mathcal{NPR}$ tuple – remember that $\mathcal{NPR}$ tuples are also indexed by their $n.\alpha_p$ field. This operation requires 2 DHT lookups every epoch $\mathcal{E}$: one lookup to delete the index tuple for the old value of $n.\alpha_p$, and one to insert the index tuple for the new value of $n.\alpha_p$. The overall cost, in terms of hops, is $O(log(N))$ (since every lookup needs $O(log(N))$ hops in Chord), while the bandwidth consumption is minimal given the very small size of these index tuples. In a Chord ring in the steady state, we can avoid the first lookup by keeping the IP and port (i.e. a link) to the node last seen storing the relevant index tuple, and checking with this node before doing the actual lookup. Moreover, the overall cost is tunable via $\mathcal{E}$, so we can trade-off $\mathcal{NPR}$ index freshness for bandwidth and hops.

### 4.2 Joining the RangeGuard

A node that is to join the RangeGuard uses its RG as the “bootstrap” node for the RangeGuard ring. The RG is responsible for retrieving the metadata of the candidate node and checking whether it is powerful enough (i.e. has served more requests than a predefined threshold) and has stayed online for long enough (based on the corresponding $\mathcal{U}$ value) to be allowed into the RangeGuard. If all prerequisites are met, the standard Chord join protocol is executed and the candidate node is promoted to the RangeGuard ring, otherwise the protocol terminates. After the node is a full member of the RG ring, it updates the $status$ field in its entry in the $\mathcal{NPR}$ relation on the lower-level ring to denote

![Figure 2. The RangeGuard architecture. RangeGuards form a second OP-Chord ring of their own, taking responsibility (using consistent hashing) for arcs (ranges) of nodes on the low-level Chord ring.](image-url)
that it is now a \( RG \) and notifies nodes in its arc of responsibility of its existence. Alternatively, this step may be left as part of the lower-level ring stabilization process.

**Cost of Joining the RangeGuard** The cost of joining the RangeGuard ring can be broken down into the following parts: (i) the cost to contact the \( RG \) responsible for the joining node (1 hop) and to send the relevant \( NPR \) tuple to the \( RG \), and (ii) the cost of the standard Chord \( join \) protocol, for the \( RG \) ring. In any case, the cost for joining the \( RG \) ring is \( O(\log(M)) \) hops overall, while the extra bandwidth consumption is minimal (given the size of the \( RG \) ring and the very small size of \( NPR \) tuples).

**4.2.1 Admission into the RangeGuard**

There are two ways for a node to be admitted into the RangeGuard: either (i) be promoted by a node already in the RangeGuard who wishes to shed some of its load, or (ii) volunteering to take up some region in the address space for which there exists no \( RG \).

i. Promotion Due to irregularities in the data or access distribution, a RangeGuard may get overworked with incoming requests. Moreover, it is possible for a region in the RangeGuard to be underpopulated (e.g. imagine the RangeGuard ring in its setup phases). In such cases, a member of the RangeGuard can ask for support from candidate RangeGuards by promoting them to \( RG \) status.

With the infrastructure described earlier, when a RangeGuard wants to promote a node to RangeGuard status for a region around a point \( p \) – i.e. the id of a node in a distance of at most \( \epsilon \) from a point \( p \) in the lower-level ring, with access count greater than \( a \) in the previous epoch, and a \( U \) value above \( u \) – it merely executes the range query:

\[
\text{select id from } NPR \\
\text{where } U > u \text{ and } a_p > a \text{ and } 0 \leq id - p < \epsilon
\]

The result set of this query will contain the IDs of candidate RangeGuards in the region of interest. It is then up to the \( RG \) who originated the query to select the best candidate, inform it of its promotion, and initiate the \( join \) protocol to add it to the RangeGuard ring.

ii. Volunteering Candidate \( RGs \) may lie in any region on the lower-level ring. For data/access distribution irregularity reasons similar to the ones urging RangeGuards to ask for support, it is possible for some regions to have such low data/access loads that the RangeGuard responsible for them has never been in need of support. This could result in large arcs on the lower-level ring being mapped to a single RangeGuard node, located many hops away on the lower-level ring from the first nodes on this arc. Although this is not an issue in the steady state, it may increase the time needed by a node on this arc to find a new \( RG \), should the current \( RG \) leave the system abnormally.

We, thus, allow candidate \( RG \) nodes on the lower-level ring to volunteer for an \( RG \) position; if a candidate \( RG \) detects a situation as the one described earlier (i.e. a large distance between itself and its \( RG \)), it can contact the latter and ask to be promoted to \( RG \) status. The \( RG \) is responsible for going through the \( \alpha_p \) and \( U \) checks outlined earlier and admitting the candidate to the RangeGuard or not.

**4.3 Leaving the RangeGuard**

Similarly, a \( RG \) may decide to leave the RangeGuard if it finds itself in a situation where its arc of responsibility becomes very small (due to candidate \( RGs \) being promoted to \( RG \) status in its vicinity), or the load it faces as an \( RG \) drops below some predefined threshold (e.g. an estimate of the load it should have, based on uniform data/access load).

Furthermore, based on locally available information, a RangeGuard may estimate the current size of the network with regard to participating nodes, and leave the \( RG \) ring if its immediate successor in the RangeGuard ring is less than some distance apart. This technique can be also used to probabilistically limit the amount of nodes participating in the RangeGuard ring, in an attempt to provide guarantees on the worst-case hop-count on the \( RG \) ring.

A \( RG \) that wishes to leave the RangeGuard ring, goes through the following steps:

1. It transfers all \( RG \)-related stored data to its successor on the \( RG \) ring.
2. It updates the \( status \) field in its entry in the \( NPR \) relation on the lower-level ring, to denote that it is no longer a RangeGuard.
3. Optionally, it notifies the nodes that link to it on the \( RG \) ring to update their links, or leaves this to be done as a part of the \( RG \) ring stabilization process.

Note that our approach uses standard Chord operations and our enhanced architecture to support range queries to set up and maintain the RangeGuard ring. With the exception of the second step, the procedure described above is the standard Chord \( leave \) protocol. Also note that the leaving \( RG \) needs not to notify nodes on its arc of responsibility on the lower-level ring of their new \( RG \), since discovery of the new \( RG \) will be done as part of the lower-level ring stabilization process.

On the other hand, there is the requirement for several RangeGuard peers with enhanced capabilities. This is not unrealistic, since many peers in real-life applications have been proven to be more powerful. With the notion and exploitation of RangeGuards we can harness this power heterogeneity of peers in order to facilitate the efficient processing of range queries.
Cost of Leaving the RangeGuard As outlined above, the cost for a node to leave the RangeGuard is equal to the cost of executing the standard Chord leave protocol for the RG ring, while data transfer is minimal due to the very small size of $NPR$ tuples and the size of the $RG$ ring.

4.4 Range Query Processing Using RangeGuards

With RangeGuards in the scene, a query $(a_i, v_{low}, a_i, v_{high})$ on attribute $a_i$ will be sent from the requesting node directly (in 1 hop) to the RG responsible for the requesting node’s data. After this point the RangeGuards assume responsibility to gather the requested information, using the OP-Chord algorithm described earlier, except that now all operations take place on the RangeGuard ring (fig. 3). With data placement on the lower ring being reflected on the RangeGuard ring, the requested index tuples will reside between $RG_l : succ'(h_l(a_i, v_{low}))$ and $RG_h : succ'(h_l(a_i, v_{high}))$, where $succ'$() is the Chord successor function for the RangeGuard ring.

This algorithm requires 1 routing hop to reach the RangeGuard ring, $O(\log(M))$ hops in the RG ring to reach $RG_l$, and as many routing hops as there are RangeGuards between $RG_l$ and $RG_h$. Moreover, the $O(\log(M))$ term can be further improved to $O(1)$, by using techniques similar to those presented in [6] or [22].

Since, there will probably be much fewer RangeGuards in the system than there are nodes, and RangeGuards are more powerful (with respect to computing capacities and network bandwidth) than the average node in the system, this algorithm is significantly more efficient than the one presented earlier. Specifically, for $5\% \times N$ RangeGuards, although the worst-case hop-count efficiency remains in $O(N)$, in now has a rather significant constant modifier (i.e. $5\%$ – or 20 times lower – hop-count).

Note that we require a mere $5\% \times N$ nodes to be powerful and altruistic in our setting; as relevant research has pointed out [1,20], we can expect an average $5\%$ of the node population in wide-scale peer-to-peer data sharing networks, such as Gnutella [3] and Mojication [13] to be powerful, altruistic nodes. Thus, by harnessing the full power of all these nodes, we can achieve even higher performance gains than those outlined above.

The Small Domain Problem Revisited The efficiency of the OP-Chord architecture drops when the cardinality of the attribute domain is much lower than the number of nodes in the network, since then successive attribute values are not mapped to successive nodes on the ring. Analysis (omitted for space reasons) has shown that this degradation is graceful when the size of the attribute value domain is more than one $\log(N)$ of the node population size, in which case we can still guarantee (on average) an $O(\log(N) + r)$ hop-count for a range query with span $r$, with the extra cost growing only logarithmically to $N/R$, $|D|$ being the size of the attribute value domain.

Using RangeGuards further alleviates the effects of the small domain problem in range query processing. With $M$ RangeGuards in the $RG$ ring, the extra cost due to nodes falling in sub-unit locations on the Chord ring is cut down to $O(\log(\frac{M}{|D|}))$ hops per unit of the range query. For an $N = 2^{20}$-node network and $M = 5\% \times N = 2^{10}$ RangeGuards, this boils down to requiring attribute value domains consisting of more than (approximately) a mere 3,300 values to guarantee the 1-hop forwarding during range query processing!

4.5 Modifications to the OP-Chord Overlay

All in all, we need to extend or alter the standard Chord system in the following fields. First we must provide appropriate protocols to allow nodes to join/leave the RangeGuard ring, while guaranteeing correct operation of the overall system, as well as a method to discover and use candidate RGs (described in detail in sec. 4.2 and 4.3). We also need to alter the query processing protocol, to utilize and harness the extra functionality offered by the RangeGuard (described in sec. 4.4). As far as routing state is concerned, we need to add 1 more entry to each node to point to the RG responsible for it. For fault-tolerance reasons and faster recovery from failing/leaving RGs, we may choose to maintain links to the next $k$ RGs. Note that this is still $O(1)$ to the number of nodes in the system.

With the extra information in the nodes’ routing tables, we also need to tweak the stabilization process to include the RG entry in the set of links to probe. Much like the standard stabilization process, nodes issues a query for its ID on the $RG$ ring. The relevant information is by design stored...
on the responsible $RG$; thus, the response to this query will originate from the $RG$ currently responsible for the arc in which the querying node is located. If the $RG$ responsible for the node has changed (e.g., due to more candidate $RG$s joining the RangeGuard), the node will get back a response from a different $RG$ and will thus update its $RG$ link. Finally, if all of a node’s $k$ $RG$ links have failed simultaneously, then the node can fall back to querying the lower-level ring for a RangeGuard (i.e., a node whose status field is set to true) in its vicinity.

## 5 Load Distribution on the RG Ring

In order to emulate the 95%-5% observation of [1] (i.e. 5% of all nodes serve 95% of all requests in the system), we can have nodes on the OP-Chord ring flip a biased coin and dispatch queries to the RangeGuard ring with a 0.95 probability, while processing queries solely on the OP-Chord ring with probability 0.05.

Apart from this, load distribution is balanced on the RangeGuard ring by (i) the load-aware join/leave protocols for the RangeGuard ring, and (ii) the data placement algorithm of OP-Chord along with the size of the $RG$ ring.

As far as join/leave is concerned, remember that RangeGuards may call for support from lower-ring candidate $RG$ nodes when overloaded, and RangeGuards may decide to leave the $RG$ ring if their load is too low or their arc of responsibility is too narrow. Moreover, in the former case, the $RG$ calling for help can choose among several candidate $RG$s, as returned from the relevant query. Since this node knows which part of its arc of responsibility causes it the most load, it can choose an appropriate candidate $RG$ to shed this very load. The above algorithm is able to provide us with the basics for having a balanced access load. As already mentioned, we expect candidate RangeGuards to be uniformly distributed on the lower-level ring. Thus, RangeGuards calling for help will have a good probability of finding a candidate RangeGuard in their arc of interest.

With respect to data placement, if the popularity of a value does not depend on its position in the attributes domain (e.g., value $v$ is not the most popular for all attributes in the system)) then having multiple attributes mapped on the same ring translates to having multiple popular items distributed to all nodes on the system. Even in the opposite case, $s_q$ of the OPHF (sec. 3.1) provides us with enough randomization in the data placement to guarantee similar results, as we shall see in sec. 6.2. Note that, for an $N$-node system and the worst-case skewed distribution (i.e., one value being selected with probability 1 and the rest with probability 0), then $N$ attributes are required to have a balanced load, under a best-case distribution of load based solely on the above facts. However, since the $RG$ ring is much smaller than the OP-Chord ring, the required number of attributes is much smaller (i.e., $M<<N$ attributes for an $M$-node $RG$ ring). Furthermore, due to this difference in the sizes of the $RG$ and the OP-Chord rings, every $RG$ node is responsible for the values assigned to multiple nodes on the OP-Chord ring, which leads to an even smoother distribution of the load on the RangeGuard ring.

Moreover, since our system is based on Chord, we can easily apply load balancing techniques developed for this DHT, such as using multiple virtual nodes per real Chord node [15], or migrating lightly-loaded nodes to locations of the Chord ring with high loads [9], to further balance the load on both lower-level and RangeGuard nodes.

## 6 Performance Evaluation

We have extended the basic Chord simulator\(^1\), adding support for index tuples, and range queries, and implementing on it our OP-Chord and RangeGuard architectures. We have chosen to test two aspects of the system: (i) the hop count efficiency of our range query processing algorithm, and (ii) the distribution of storage requirements and accesses on participating $RG$ nodes during range query processing, under real-world skewed distributions.

### 6.1 Hop Count

The experiments used a single-index-attribute relation, with the index attribute taking 5,000 integer values, following a Zipf distribution with $\theta = 0.7$ [20] over $D$. Range queries are generated using a separate Zipf distribution over the domain $D$ (again with $\theta = 0.7$) for their lower bound, and a uniformly distributed range span $S$, ranging from 1% to 50% of the attribute domain. We report on a series of 50,000-queries experiments for a system with $N = 1,000$ nodes, $M = 50$ ($\approx 5\% \times N$) range guards, and 50,000 tuples (the reported results are not sensitive to these values). The maximum size of the nodes’ finger tables is set to $\log(N)$ entries ($N$ being the number of nodes in the system, known at simulation time).

#### Performance Reference Points

We have compared the hop-count efficiency of the RangeGuard ($RG$) architecture against (i) plain Chord (PC), (ii) an imaginary, enhanced Chord (EC), where for each range $R$ the system knows the IDs of the $n'$ nodes storing values in $Ra$, (iii) our OP-Chord architecture, and (iv) a hybrid system where 95% of queries are processed on the $RG$ ring and the remaining 5% are dealt with on the OP-Chord ring (as discussed in sec. 5).

Assume we have an integer range query $r = <v_{low}, v_{high}>$. Further assume that the requested index tuples are stored on $n'$ nodes, under Chord’s hashing scheme, and on

\(^1\)Available through http://www.pdos.lcs.mit.edu/chord/
$k$ nodes, managed by $k'$ RangeGuards, under our OPHF scheme. Then:

- $PC$: $|r|$ queries are needed to gather all possible results, for an overall hop count of $O(|r| \log(N))$.

- $EC$: $|r'|$ queries must be executed to gather all possible results, for an $O(|r'| \log(N))$ overall hop count.

- $OP$: we must first route to the node holding $v_{low}$ and then follow $k-1$ successor pointers to gather all possible results, for an $O(\log(N)+k)$ overall hop count.

- $RG$: we must route to the closest $RG$ (1 hop), then route to the $RG$ holding $v_{low}$ ($O(\log(M)$ hops) and then follow $k'-1$ successor pointers to gather all possible results, for an $O(\log(M)+k')$ overall hop count.

Results Fig. 4 summarizes the measured hop-counts per range query. Chord wasn’t designed with range queries in mind (used here only as a reference point) thus it performs poorly. The (unrealistically) enhanced Chord brings the required hop count down to $\approx20\%$ of the Chord figure. However, total global knowledge is required to implement this approach – every node must know the exact distribution of data items on nodes and whether a certain data item exists in the network or not. On the other hand, by using our order-preserving hashing scheme and the RangeGuard architecture, the hop count is decreased by a factor of approx. 50 to 500 compared to $PC$, 10 to 110 compared to $EC$, and 5 to 20 compared to $OP$ for different range spans, with the performance of $OP + RG$ following closely behind.

6.2 Load Distribution and the RangeGuard

The effect of $s_0$ and of overlapping multiple attributes in the access/storage load balancing is beneficial in our setting. To showcase this claim, we have performed the following experiment: assume we have an OP-Chord ring with 14-bit IDs. We add nodes to the system, at random positions on the OP-Chord ring (simulating the quasi-uniform placement resulting from the use of SHA-1). We let the system stabilize and add 20,000 multi-attribute tuples in the system. The values of the index attributes are drawn from the $[1, 40, 000]$ integer interval according to a Zipf distribution with $\theta = 0.7$. If, on the other hand, we assume a uniform value occurrence distribution (as opposed to the above Zipfian distribution), the following results carry on to a Zipf load access distribution.

We vary (i) the number of index attributes per tuple, from 1 attribute (the classic single-attribute case of the currently available Chord system) up to 400 attributes, and (ii) the number of nodes in the network from 1,000 to 5,000, 10,000, and 20,000. Note that, e.g. in the 20,000-node case, should these nodes be $RG$ nodes, they would be enough to administer a 400,000-node network, under the 5%-intuition described earlier.

Figure 5 shows the ratio of the highest to the lowest load in the system. Naturally, the optimal load ratio is 1, in which case all nodes in the system will have the same load. With a $\theta = 0.7$ Zipfian value occurrence distribution in an 1,000-node network, the highest-to-lowest single-attribute node access/storage load ratio load is 7.5, dropping to 1.97 for 8, and 1.06 for 400 attributes. We have noted on the figures the load ratio for the single-attribute case (denoted by the “load = ” points) and the number of attributes required for this load to drop below 2 (denoted by the “# of attributes” points). With nodes being placed on the lower-level ring using Chord’s SHA-1, we can expect RangeGuards to be uniformly distributed on the ring. Thus, the above situation holds for both the lower-level ring and the RangeGuard ring of our architecture. Note, however, the increase in the latter with the number of nodes in the network. As we expect $RG$ nodes to be a small percentage of all network nodes, the above results show that, for the $RG$ ring, load distribution will be within acceptable bounds (even without the other relevant mechanisms discussed in sec. 4). For much larger networks, we shall either need a very large number
of attributes to achieve a good load distribution and/or more elaborate load balancing mechanisms [14].

7 Discussion

The driving force for this work is based on two main points: (i) we believe that, in order to achieve better performance figures and break the current hop-count barriers in P2P/DHT query processing, we must harness the extra horsepower of powerful and altruistic nodes, proved to exist in widely-deployed P2P sharing networks, and (ii) when it comes to range queries, this demand becomes even stronger, since the cost of range queries is not just the cost to get to one node (e.g. the first node storing the lower value of the range) as in exact-match DHT lookups, but also to gather all values in the requested range. The validity of the latter has been proven by our analysis and experimental performance results for the Chord and Enhanced Chord cases.

As far as the first point is concerned, relevant research [1, 20] has already proven that such nodes exist in currently deployed P2P sharing networks, such as Gnutella [3] and Mojonation [13]. To be more precise, [1] have shown that about 5% of the nodes in the Gnutella network serve over 95% of all queries circulating in the system. What we propose is to harness the processing horsepower, storage space, and network bandwidth of such nodes in a distributed coordinated way, in order to achieve high performance in range query processing. Moreover, it should be noted that in addition to the significant average hop-count improvements offered by the RangeGuard architecture, there are important qualitative factors that should be considered, stemming from the fact that query processing is performed by powerful nodes. For example, if RG nodes have a network bandwidth that is $x$ times greater than average, then overall performance is further improved by this $x$ factor.

Finally, the RangeGuard could be used just to speed-up the normal location/routing primitives of the underlying DHT, by using techniques described by [6] and [22], we can route from a source node to a target node by sending the message to the source node’s $RG$ in $O(1)$ hops, having it routed in the $RG$ ring in $O(1)$ hops to the $RG$ responsible for the target node, and from there in another $O(\log(5\% \times N))$ hops with $O(5\% \times N)$ RangeGuards, or $O(\frac{N}{\log(N)})$ hops with $O(\log(N))$ RangeGuards to the target node; with $M$ RangeGuards, the lower-level ring is divided in partitions of $\frac{N}{M}$ nodes each, hence the above limits.

8 Related Work

Gribble et al. [4] first investigated how P2P systems can gain the strengths of data management systems, which could enable them to provide complex queries capabilities. [8] and [23] have also looked into supporting complex queries upon DHT infrastructures. [8], however, did not address range queries. Gupta et. al [5] propose an architecture based on Chord, and a hashing method based on a min-wise independent permutation hash function. However, they provide only approximate answers to range queries. Sahin, et al. [19] extend the CAN DHT for $d=2$ dimensions, to allow for range query processing. However, for $d=2$, CAN performance is considerably inferior compared to the other DHT-based solutions. Andrzejak and Xu [2] also propose a CAN-based extension for range querying, thus suffering from similar shortcomings (i.e. higher hop counts and less robustness/fault-tolerance, compared to Chord).

Triantafillou and Pitoura [23] first presented the idea of using an order-preserving hash function on Chord and provided a preliminary algorithm for exploiting this placement for range query processing. Compared to [23] (the overlap of which with the present paper is a subset of the text in sections 3.1 and 3.2) we have extended the OP-Chord architecture with (i) more efficient algorithms for range query processing, (ii) a discussion and analysis of the relevant small-domain problem, and (iii) with the RangeGuard architecture and its performance study.

Karger and Ruhl [9] have presented a load balancing solution based on relaxing some of the constraints of Chord’s consistent hashing scheme: first, allowing every node to take one of $O(\log(N))$ positions on the Chord ring, defined as hashes of the node ID), and then completely decoupling the placement of nodes on the ring from any verifiable information (such as the SHA-1 hash of the node’s IP address and port number, used in the standard Chord system). This latter decision is core to their approach to range query processing. On the other hand, our OP-Chord/RangeGuard architecture keeps the secure-hash-based placement of nodes on the ring, while providing for efficient range query processing and storage/access load balancing. Furthermore, we expect that in most application environments there will be several relations, each with several attributes, being stored in the P2P data network. This, as our study (sec. 6.2) has shown, is enough to ensure good load distribution characteristics among the nodes of the network (a companion paper [14] addresses, in general, the issue of storage and access load balancing in concert with the OP-Chord architecture for efficient range query processing). Finally, as pointed out in [17], appropriately using powerful nodes in the location/routing primitives of DHTs, can lead to both a more scalable [11] and more efficient system.

All earlier research efforts failed to recognize and exploit the key fact that the appropriate utilization of such powerful nodes can speed up query processing significantly. Viewed from a complementary angle, earlier research failed to recognize that in large scale data sharing networks, for example, there exist nodes which are weak, with respect to their processing, storage, and communication capacity
(i.e., nodes with intermittent connectivity and/or low bandwidth), and that there also exist nodes with orders of magnitude more horsepower [20]. Our proposal avoids the pitfall of relying upon weak nodes for query processing. Furthermore, we follow a data management approach to discovering and harnessing powerful nodes: keeping metadata for participating nodes as a relation on our OP-Chord architecture allows us to swiftly and efficiently locate such nodes and, by promoting them to RangeGuard status, to use them in the core of routing and query processing.

9 Conclusions

With this work we address the problem of efficient range query processing in structured P2P networks. Our approach leverages existing DHT-based P2P research. Our approach is based on: (i) first, extending the basic Chord architecture, using an order-preserving hash function and related algorithms for tuple addition / deletion and routing requests (query processing for range queries), and (ii) second, a new architecture that facilitates the exploitation of powerful nodes, coined RangeGuards, in the network, assigning to them specific tasks for further significant speedups during range query processing. This architecture is based on: (i) adding a second order-preserving Chord ring, (ii) a way to identify and collect RangeGuards, and (iii) mechanisms to utilize them during range query processing.

Our proposal supporting range queries enjoys significant performance advantages compared to related work. Our analysis and experiments show that the proposed OP-Chord and RangeGuard architectures can significantly improve the efficiency of range query processing in terms of average hop counts. Furthermore, our architectures can achieve good storage and access load distribution. Finally, and perhaps most importantly, a key advantage of the proposed architectures is that they avoid the dangers and inefficiencies of relying on weak nodes for range query processing, with respect to their processing, storage, and communication capacities, and their intermittent connectivity.

For the future, we plan to investigate the applicability of randomized algorithms (e.g., random walks) for the detection of powerfull nodes and their addition to the RangeGuard. Furthermore, we intend to explore the potential of using multiple RangeGuard rings. For example, we could have one RG ring for every index attribute, to speed-up multi-attribute range query processing. Alternatively, we could dynamically add RG rings, when the load of the current rings gets too high, have RGs migrating from one ring to another based on the load characteristics of these rings, and have lower-level nodes randomly choose one of these rings to execute queries. Such a setup could lead to better load balancing within the RG community; however, there are many open issues to be examined.

References