Document downloaded from:

https://riunet.upv.es/handle/10251/232971

This paper must be cited as:

Hidalgo, N.; Rosas-Olivos, Erika Susana; Arantes, L.; Marin, O.; Sens, P.; Bonnaire, X. (2012). Optimized Range Queries for Large Scale Networks. En IEEE Computer Society, AINA 2012 (pp. 438-445). https://doi.org/10.1109/AINA.2012.32

The final publication is available at

http://dx.doi.org/10.1109/AINA.2012.32

Copyright IEEE Computer Society

Additional Information

Optimized range queries for large scale networks

Nicolas Hidalgo*, Erika Rosas*, Luciana Arantes*, Olivier Marin*, Pierre Sens* and Xavier Bonnaire†
*Université Pierre et Marie Curie, CNRS - INRIA - REGAL, Paris, France
E-mail: [firstname.lastname]@lip6.fr

†Department of Computer Science, Universidad Técnica Federico Santa María, Valparaíso, Chile
E-mail:firstname.lastname@inf.utfsm.cl

***Abstract*—Distributed Hash Tables (DHTs) provide the substrate to build scalable and efficient Peer-to-Peer (P2P) networks which are distributed systems with the potential to handle massive amounts of data on a very large scale. However, traditional DHTs provide very poor support for range queries.**

In this article we present a search mechanism that efficiently supports range queries over a ring-like DHT structure using a prefix tree index. Load balancing is improved by delegating the routing of queries to the nodes that store data, and by updating neighbor information through an optimistic approach. Our solution produces low overhead and low latency in environments where queries outnumber data insertion operations. We analyze the performance of the system through simulations on top of Peersim and show that our solution does not rely on data distribution.

***Keywords*-Peer-to-Peer, DHT, Information Retrieval, Range Queries.**

I. Introduction

Supporting range queries in large-scale, dynamic networks and acquiring an exhaustive list of results in a timely manner is now a common issue for a wide variety of distributed applications such as music/movie storage, peer-to-peer persistent games, scientific computation, data mining, and many types of data warehouses. A range query retrieves all the objects with values within a given range.

Peer-to-peer (P2P) systems offer an abundance of resources, with a higher data storage potential [**?**]. They are autonomous, fully decentralized, self-organized and highly scalable with the potential to grow up to millions of nodes. Furthermore, Distributed Hash Tables (DHTs) built over P2P are considered to provide efficient lookup functionality. They are scalable, fault tolerant, and offer load balancing. Among many, some well-known DHT-based overlays found in the literature are Pastry [**?**], Chord [**?**], and Kademlia [**?**] on top of which a lookup operation is performed in $\mathcal{O}(\log(N))$ hops, where N is the total number of nodes of the network. However, DHTs do not support range queries efficiently since the use of uniform hashing destroys data locality.

Several solutions in the literature attempt to provide efficient support of range queries over DHT-based P2P networks. However, none of them manage to combine load balancing, low message overhead, and low search latency simultaneously while preserving the scalability of the system. The existing solutions mainly fall into two categories: *over-DHT* and *overlay-Dependent* solutions. *Over-DHT* approaches build an additional structure or index, which preserves data locality an thus allows to support range queries [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. Several of them rely in tree-based indexing structure (e.g., PHT [**?**], RST [**?**], DST [**?**], and LIGHT [**?**]). On one hand, as *over-DHT* solutions are built over a generic DHT, they preserve the good properties of the latter, are portable, and are easy to implement. On the other hand, their main drawbacks lie in the extra maintenance cost, in the significant query latency due to the top down nature of traversal searches over the structure, and in the load balancing problems associated to the search process. *Overlay-Dependent* approaches directly support range queries by adapting the DHT (e.g. Yarks [**?**], Skipnet [**?**],Skipgraph [**?**], Hivory [**?**]), by using non-uniform mapping techniques (e.g. locality-sensitive hashing (LSH), SHA-1) which preserve data locality (e.g. [**?**], MANN [**?**], Mercury [**?**]) or by adopting a tree-based approach (e.g. Pgrid [**?**] (trie structure), BATON [**?**] (AVL tree), Skip tree graph [**?**] (skip tree), Hyperring [**?**] (deterministic 2-3 trees), Ptree [**?**] (B+ tree), among others). Nevertheless, *overlay-Dependent* solutions present load balancing and portability problems while some of them induce high latency and substantial maintenance costs.

In this paper we aim at improving range queries search in environments where queries significantly outnumber data insertion operations; resource discovery and data mining belong to this category of applications. We thus propose a solution which focuses on the search process; it works independently of the data distribution and can perform range queries in the order of $\mathcal{O}(\log(L))$ steps, where L is the number of nodes that store data. Based on the indexing structure of Prefix Hash Tree (PHT) [**?**], our approach provides a high performance range query management: it reduces search latency, achieves good load balancing among the nodes, and produces low message traffic overhead by diverting the queries to the nodes that store the data. We conducted an extensive set of simulations which demonstrates a significant improvement when compared to the original PHT approach with a reduction of traffic overhead by at least 65%.

The rest of this paper is organized as follows. Section **??** briefly describes the overlay and the indexing approach on top of which we build our solution, while Section **??** presents the structure and search operations. In Section **??** we present a performance evaluation conducted through simulation on top of Peersim [**?**]. Finally, Section **??** concludes this paper.

II. Background

A DHT is a self-organizing structured substrate built over peer-to-peer overlay networks in which any data can be located within a bounded number of routing hops. Several existing DHT overlays like Chord [**?**], Pastry [**?**] or Tapestry [**?**] logically organize the nodeID space of nodes into a ring.

Our work uses Pastry [**?**], a ring-like structured DHT. Every node in Pastry is associated to a unique nodeID in an m-bit space using the hash function SHA-1. Numeric keys represent application data (objects) and are chosen from the same name-space as the node identifiers. Pastry assigns thus each key k to the node whose nodeID is numerically closest to k.

While lookup operations can be efficiently executed over DHTs, the latter do not provide support for complex queries, such as range queries. To this end, several indexing schemes have been proposed in the literature [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. We base our work on Prefix Hash Tree (PHT) [**?**], which provides an indexing data structure built over DHT-based P2P networks.

In PHT, keys of objects to be indexed are within the domain $\{0,1\}^D$, where D is the length of the string. PHT is thus a binary prefix tree (binary trie) where the left branch of a node is labeled 0 and the right branch is labeled 1. Each node n of the trie is identified with a chain of P bits (prefix) produced by the concatenation of the labels of all branches in the path from the root to n. PHT builds a prefix tree in which objects are stored at *leaf nodes*. Hence, an object with key k is stored at a leaf node whose label is a prefix of k. Fig. **??** illustrates an example of a PHT.

Compared to other tree-linked solutions [**?**], [**?**] where a B-tree with O objects requires $\log(O)$ lookups, PHT requires only $\log(D)$ lookups, where D is the key size. Additionally, replication over the DHT can avoid data loss.

PHT trie is completely distributed among the peers of the network by *hashing* the prefix labels of the PHT nodes over the underlying DHT identifier space. We call *internal nodes* those nodes that belong to the PHT trie but are not leaves while those nodes that do not participate to the PHT trie we denote *external nodes*.

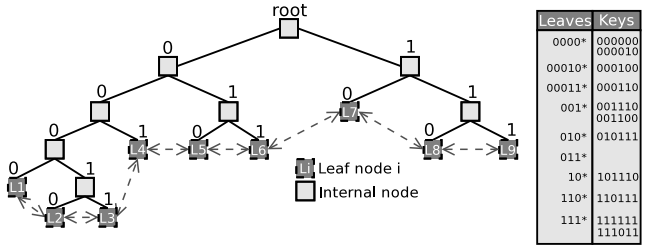


Fig. 1. PHT indexing structure.

Whenever a leaf node X reaches its maximum storage capacity, it splits into two children. The left child will have the prefix label of its parent concatenated with 0 and the right child with 1. Furthermore, the information stored on X is distributed among the two children.

Classic searches in PHT follow either a linear or a binary method. A linear search starts with the node that corresponds to the smallest possible prefix of a given key k. The next step consists in a DHT lookup operation in order to find the next node in the trie whose prefix is one bit more similar to k. The lookup operation is repeated until a leaf node is reached. This search method produces a number of *DHT-lookups* of order D. A binary search is a half-interval process that starts by querying a middle prefix of D. If the prefix corresponds to an internal node of the PHT, the search discards the lower half of the interval and continues querying a new middle prefix of the remaining interval. If the prefix corresponds to an external node, the search discards the upper half of the interval. This search method produces a number of *DHT-lookups* of order $\log(D)$. However, in dynamic scenarios the binary search can fail, i.e. be unable to correctly locate the leaf node [**?**].

PHT maintains a double list which links all leaf nodes (*Threaded leaves*), as shown in Fig. **??** by the dashed lines. This list allows to search all data of the requested range sequentially, starting from the lower bound. The sequential scan of the list can be avoided by parallelizing the search within the range. PHT [**?**] proposes to start from the node that corresponds to the prefix that completely covers the requested range and then continue making a number of searches that correspond to the number of children which overlap the range, until reaching all the leaf nodes. However, if the trie is highly unbalanced, the number of messages will grow without decreasing the latency when retrieving all data. In Section **??**, we refer to this type of search as *PHT-parallel*.

III. Optimizing Range Queries

This section details PORQUE (Peer Optimized Range QUEries), a two-layer ring structure that effectively supports range queries. A node exploits information provided by range queries that it has previously issued. When performing a new query, such information is used to contact the leaf nodes directly, thus reducing search latency. In order to further optimize search latency, we take into account range coverage.

A. Building a Double-ring Architecture

PHT nodes are uniformly distributed among the nodes of the DHT, based on the prefix of their key. In order to improve the performance of range queries, our approach maintains a second ring structure over the DHT overlay. This second ring, denoted *Leaf Ring*, has a structure based on Skipnet [**?**] and comprises only the *leaf nodes*, i.e. nodes that store data. The identifier of a node in the Leaf Ring is its corresponding label identifier in the trie. Such a choice improves sequential data search, and therefore facilitates range querying.

In the Leaf Ring, prefix labels of leaf nodes are ordered organized into a ring, i.e., each node has a reference both to its successor and predecessor leaf in the trie, equivalent to the double list structure of PHT. Notice that PHT exploits this list for data retrieval only, not for searching. Fig. **??** presents the double-ring proposed architecture, where nodes in grey represent leaf nodes.

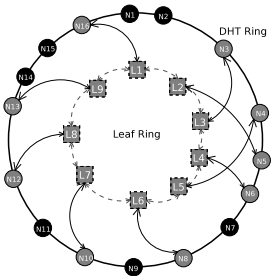


Fig. 2. Double structure: Leaf Ring over the overlay ring

In addition to successor and predecessor references, every node in the *Leaf Ring* maintains a *Search Table* which stores references to other leaf nodes in the *Leaf Ring*. Fig. **??** shows an example of a Search Table. The maximum number of entries is the maximum object key size D. The ith row of the Search Table contains two references to nodes that are at a *distance* of 2^i hops in both directions (forward and backward) to a prefix label identifier. For example, for node $L5$ in Fig. **??**, the nodes that are 2 (2^1) hops forward and backward are $L7$ and $L3$ respectively.

Each reference consists in the prefix label of a leaf node, as well as a static reference (IP Address) to this leaf node. Maintaining the prefix label helps in dynamic environments, where static references become obsolete very fast. When the access to a static reference fails, PORQUE contacts the leaf node by hashing the corresponding prefix label and looking it up in the DHT.

In order to effectively access the Leaf Ring structure, information from past queries is stored on every node in the P2P network. This information allows any DHT node to contact the Leaf Ring directly, thus avoiding the internal level of the PHT trie-structure (see Section **??**).

Within the Leaf Ring layer, searching follows the routing technique of the underlying DHT structure: using a greedy technique that forwards the search to the node which is closest to the key.

SEARCH TABLE
Node L5

	Distance from L5	Fwd_Prefix	Fwd_IP Address	Back_Prefix	Back_IP Address
1	2^1	10	190.168.0.15:99	00011	200.50.0.15:99
2	2^2	111	200.1.19.85:99	0000	81.168.0.15:99
3	2^3	001	81.160.5.1:99	011	132.227.64.15:99
⋮					
i	2^i	...	...	...	...

Fig. 3. A Search Table of a node

A node iteratively fills its Search Table the first time it joins the Leaf Ring. To complete the forward reference at distance 2^{i+1} in the Search Table, a node X asks the node at distance 2^i for the ith forward reference in its table. For example, to find the node at forward distance 2^1, X can obtain the information from its successor (since it is at distance 2^0); and to find the node at distance 2^5, X can ask the node at distance 2^4 to return the 4th entry of its Search table. The same goes for backward references, with the predecessor set as the node at a backward distance of 1.

The part of our approach described above performs efficiently if the trie is balanced. Indeed, join and leave operations induce a global change in the tables of the leaf nodes, which grows linearly with the number of nodes. This results from taking into account distances instead of making a static partition of the name-space. It is worth noticing that a static partition of the index space is not efficient either: when the trie is unbalanced, the load distribution follows the disequilibrium. Therefore, in order to overcome this balancing problem and repair the Leaf Tables of nodes, we propose an optimistic approach which consists in updating tables only when performing a range query (see Section **??**). We argue that this suffices to achieve good performance in scenarios where queries significantly outnumber data insertion operations.

B. Split Operation

Every split operation in the logical PHT trie induces two joins and one leave operation in the Leaf Ring. The split node is removed from the Leaf Ring and both its children join the Leaf Ring.

Whenever a node splits, this node – denoted parent – must inform its children about the identifier of its predecessor and successor nodes in the Leaf Ring: the predecessor of the left child is its parent's predecessor in the Leaf Ring and its successor is its right brother; similarly, the predecessor of the right child is its left brother and the successor is its parent's successor in the Leaf Ring. The children Search Tables are thus initialized with the information provided by their parent node and they are further updated in an optimistic way as described in Section **??**. Algorithm **??** presents how a split operation is performed.

Fig. **??** shows an example of a split operation in the Leaf Ring. The node with identifier 10 reaches its storage capacity and is replaced by two other nodes with prefixes 100 and 101. The information from the node with prefix 10 is used by its children in order to update the successor and predecessor links as well as the Search Table.

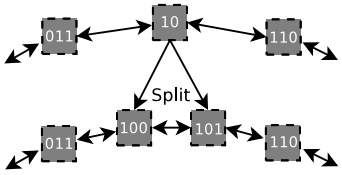


Fig. 4. A split operation in the Leaf Ring

In case of concurrent splits of neighbor nodes of the Leaf Ring, the obtained information that a child gets from its parent might be out of date. Thus if the parent's successor, Y, is no longer in the Leaf Ring, the right child will contact the left child of Y. Notice that the prefix label of Y's left child can be obtained by the concatenation of Y's prefix label with '0'; similarly, if the parent's predecessor, Z is no longer in the Leaf Ring, the left child will contact Z's right child. The nodeID

of the latter is composed by the concatenation of Z's prefix label with '1'.

Algorithm 1: Split Operation

```
input : Node parent; /* Splitting node*/
/* Children ids */
left_id = parent.id.append(0);
right_id = parent.id.append(1);

left_child = sendLookup(left_id);
right_child = sendLookup(right_id);

/* Transferring data */
if commonPrefix (left_id, data.prefix) == data.prefix then
    sendData (left_child);
else
    sendData (right_child);
join (left_child, parent.predecessor, right_id,);
join (right_child, parent.successor, left_nodeId);
/* Nodes Join(x, y, z). x: Node to contact, y:
Predecessor z: Successor */
```

C. Optimistic Table Maintenance

We propose to repair the Search Tables optimistically: reference updates only occur when a range query takes place.

In Pastry [**?**], data corresponding to a leaf node with prefix label A is stored on the node whose identifier is closest to $SHA(A)$. Since DHT nodes can crash or leave the network, Pastry replicates data on several nodes in the numerical vicinity of $SHA(A)$ so as to avoid information loss. Therefore in dynamic environments, the node responsible for another node can change. When a leaf node leaves the network or crashes, its static reference is no longer valid. In this case, Pastry falls back on the new current node whose identifier is numerically closest to $SHA(A)$ as the holder of the data. Given the prefix label, this new node can easily be contacted through a DHT lookup operation and the static reference to its IP address is updated. It is worth pointing out that no other static reference in the Search Table requires an update.

Algorithm 2: Maintenance Operation (`maintenance`)

```
input : Entry e
if e != null then
    /* maintenance check*/
    if e.type == "forward" then
        if e != getEntryForward (e.index) then
            reply (getEntryForward (e.index - 1));
    if e.type == "backward" then
        if e != getEntryBackward (e.index) then
            reply (getEntryBackward (e.index - 1));
```

Every split of a leaf node of the trie produces inconsistency in its Search Table. Similarly, this table is only updated when a range query takes place. As will be described in Section **??**, a range query is forwarded to a leaf node which is numerically closest to one of the bounds of the range query. Upon receiving the query, this node looks into its Search Table references and forwards a message to the i reference which is numerically closest to the searched node, also including its table reference $i + 1$ in the message. The node that receives the message checks if this reference corresponds to its own at row i in its table. If such is not the case, it passes on its entry reference to the sender which will then update its table. The maintenance operation is presented in Algorithm **??**.

Since queries are necessary to keep the Search Table updated, PORQUE performs optimally when range queries outnumber insertions of data significantly. In the opposite case, the nodes in PORQUE cannot update their tables efficiently enough and the performance diminishes (see Section **??** for details). If there are one or n splits of data, the number of maintenance messages will be the same: one for each query transmission. If a query reaches its target in d hops, PORQUE may generate d maintenance messages if every entry is out of date. Since these messages are at the leaf level and they are one-hop messages, the impact on performance remains low.

D. Range Query Strategies over the Leaf Ring

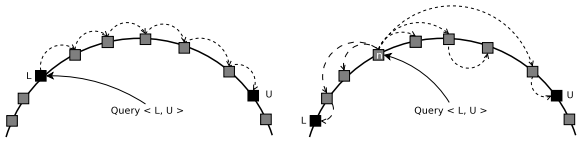
Every node in the DHT maintains a *Cache List* which contains the leaf nodes *contacted* during previous range query requests. *Contact nodes* are thus potential starting points for searches over the Leaf Ring, i.e. the nodes which provide direct access to the Leaf Ring.

The size of the Cache List is a parameter of the system. Every entry stores both the static IP address reference and the prefix label that identifies a leaf node in the trie (and in the Leaf Ring). If an access first fails, a DHT lookup operation finds the new contact node by applying the hash function over the prefix label kept by the entry. When the Cache List is full, new entries replace old entries following a Least Recently Used (LRU) strategy.

If a node has never performed any query yet, its Cache List is empty: it knows no *contact node*. In order to fill its Cache List, a node first asks for the Cache List of its neighbors. However, if such an information is not available either, it applies one of the PHT search mechanisms (linear or binary) [**?**], as explained in Section **??**.

A range query corresponds to an interval of data to be searched. It has the form $R_q = [l, u]$, where l is the lower bound and u is the upper bound. To collect the data in the range, we propose two search protocols:

- *Sequential Search*: The simplest mechanism is to start the search at one of the bounds. Therefore, the requester must select a node in its Cache List (*contact node*) with a prefix label numerically close either to the lower or to the upper bound as is presented in Algorithm **??**. The search will start from the bound value that has the greatest common prefix label with the nodes referenced in the Cache List of the requester. If both bound values satisfy this condition or if the Cache List is empty, either of the bound values can work as the *starting search node*. From the contact node, the target bound can be easily found through the Leaf Ring. The routing process followed to retrieve the query information is presented in Algorithm **??**. Firstly, the contact node must check if it is in charge of the query target by invoking the function `imClosest`. If it is the case, the `getRange` sequential



(a) Sequential Search (b) Optimized Search

Fig. 5. Search Schemes

function is invoked to start the retrieval process. To retrieve all the data comprised in the range query, the search can continue along the successor or predecessor links of the Leaf Ring. The `getRange` sequential is presented in Algorithm **??**. In the other hand, the query is routed to the closest node including entry $i+1$ of its Search Table. The routing direction depends on which node provides the shortest path for the query target. Fig. **??** presents an example where the lower bound is the starting point of the range search.

- *Optimized Search*: In order to reduce search latency, the information of the entries of the Search Table allows concurrent access to the leaf nodes that contain data within the range of the query. In this case, the starting search node is the one that stores the keys associated to $(l+u)/2$, i.e. the node in the middle of the requested range (Algorithm **??**). However, as the routing forwards the query to the node which is numerically closest to this value, it is highly possible that the query will reach a node within the range before it reaches the middle of the requested range. The routing process is presented in Algorithm **??**. Note that in this case the function `imClosest` have a different meaning than the one used in the sequential approach. In the optimized case, for any node within the lower and upper bound the function will return true. Therefore, when the request arrives at any leaf node contained in the searched range, this node forwards/backwards the search messages to all the nodes of its Search Table able to answer the query. All receiving nodes will apply this same approach. The retrieval process is presented in Algorithm **??**.
  So as to avoid redundant forwarding/backwarding messages, the query message collects information about previously visited leaf nodes. Fig. **??** illustrates this scheme. Node n, the initial receiver of the query, forwards the query to its successor and two other nodes of its Search Table. In parallel, it also backwards it to its predecessor and one other node of its Search Table. All these nodes have a prefix that covers part of the range query.

Our approach also supports other complex queries in a simple, efficient way. For instance *Min/max queries* look for the smallest and largest value of the indexed data. PORQUE carries them out with a single lookup operation. *K-NN queries* return the k nearest data values to a given key. The successor and predecessor links of the Leaf Ring allow to find the required data efficiently.

Algorithm 3: New Query: Sequential Search

```
input : Query Q
n1 = cacheList.getClosest(Q.lowerBound);
n2 = cacheList.getClosest(Q.upperBound);

if (Q.lowerBound - n1.id) < (Q.upperBound - n2.id) then
    Q.target = Q.lowerBound;
    route (Q, n1, null, null );
else
    Q.target = Q.upperBound;
    route (Q, n2, null, null );
```

Algorithm 4: New Query: Optimized Search

```
input : Query Q
Q.target = (Q.lowerBound - Q.upperBound) / 2;
nextHop = cacheList.getClosest(Q.target);

route (Q, nextHop, null, null );
```

Algorithm 5: Route Operation

```
input : Query Q, Entry e
/* Locating next hop and entry*/

n1 = getClosestBackward (Q.target);
n2 = getClosestForward (Q.target);

/* Functions imClosest and getRange depends of the search
protocol: Sequential or Optimized */
if imClosest then
    getRange (Q);
else
    if (Q.target - n1.id) < (Q.target - n2.id) then
        route (Q, n1, getEntryBackward ( i+1) );
        /* forward query Q backward to n1 including entry
        i+1*/
    else
        route (Q, n2, getEntryForward ( i+1));
        /* forward query Q forward to n2 including entry
        i+1*/
maintenance (e);
```

Algorithm 6: Get Range: Sequential Search

```
input : Query Q, Entry e
sendData ();
if Q.target == Q.lowerBound then
    if this.successor =< Q.upperBound then
        getRange (Q, this.successor, getEntryForward(1));
else
    if this.predecessor => Q.lowerBound then
        getRange (Q, this.predecessor, getEntryBackward(1) );
maintenance (e);
```

Algorithm 7: Get Range: Optimized Search

```
input : Query Q, Entry bound, int dir
/* dir: -1 if backward, 1 if forward and 0 if both search
directions*/
sendData ();
foreach Entry e in Search Table do
    if e.fwdPrefix != null and dir >= 0 then
        if e.fwdPrefix < bound and e.fwdPrefix < Q.upperBound then
            getRange (Q, e.fwdAddress, 1,
            next_fwd_entry_in_table );
    else if e.bwdPrefix != null and dir <= 0 then
        if e.bwdPrefix > bound and e.bwdPrefix > Q.lowerBound then
            getRange (Q, e.bwdAddress, -1,
            next_bwd_entry_in_table );
```

IV. Performance Evaluation

In this section we discuss performance evaluation results when using PORQUE and PHT to perform queries. Our experiments were conducted on top of Peersim [**?**] using an event-driven model.

Queries are generated over data sets with different distributions: Uniform, Gaussian, and the real dataset of DBLP [1]. The indexed attribute in DBLP dataset is the author's name. Nodes that generated queries are selected randomly.

We consider a network of $10,000$ peers, where each peer can store at most 100 objects. During simulation we perform a total of $50,000$ queries, and take a snapshot of the simulation every $1,000$ queries. The range span of the queries is the minimal for the first experiments in order to give a separated analysisWe present our most interesting results in Fig. **??**//.

Data distributions: The trie structure shape is tightly related to the indexed data keys. Uniform distributions generate uniform tries where all leaf nodes are located in almost the same level. On the other hand, skewed distributions, as Gaussian and DBLP, generate unbalanced tries where clustered data, i.e., frequent data keys within a given range, are stored in the deeper leaf nodes of the trie.

Gaussian and DBLP dataset present skewed data distributions where keys are clustered on small part of the whole key range. DBLP dataset has a different distribution where keys are clustered in many spots. A comparison of both Gaussian and DBLP datasets is given in Fig. **??**.

Traffic Message Overhead: We have compared the total number of messages generated in the search processes. Furthermore, each simulation starts by inserting $20,000$ objects in the network. In the case of PORQUE the results also include the maintenance messages, which actually represent only 0.1% of the total traffic.

Fig. **??**, **??**, and **??** show the message traffic for Uniform, Gaussian, and DBLP dataset distribution respectively. These figures illustrate how strongly the performance of PHT searches depends on the distribution of data, which defines the shape of the index trie. Linear PHT generates more messages on deep tries. This occurs in the case of DBLP (Fig. **??**), where data distribution is skewed.

[1]DBLP Dataset http://dblp.uni-trier.de/xml/

The traffic associated to the binary PHT search also depends on the shape of the index trie structure: for the Gaussian and DBLP distributions, PHT binary search can find a leaf node very fast. We should point out that this result relies heavily on the maximum key prefix size (which in our experiments is set to 40) and on the size of the searched branch.

Comparatively, the number of messages generated by our PORQUE search process is far smaller than for both PHT searches. In addition, PORQUE search performance does not depend on data distribution, as it avoids access to internal nodes of the indexing structure and performs the search only over the leaf nodes. At the beginning of the experiment peers do not know any leaf node, so nodes will use the default PHT linear search. Every node quickly starts adding leaf nodes to its Cache List as it handles queries. After 10,000 queries PHT search is not longer used, and the beneficial impact of PORQUE in terms of messages becomes obvious. In the case of the Uniform distribution, the number of messages is reduced by over 80% and the improvement reaches almost 70% for the Gaussian and DBLP distributions.

Latency: In highly dynamic environments exhibiting significant churn, static references are useless and DHT-lookup allows to find data. Therefore, in order to assess latency, we measure the number of hops necessary to reach a leaf node during DHT-lookup operations. We then compare the results for PHT and PORQUE searches.

Fig. **??**, **??**, and **??** show the number of hops relative to the search process for all three distributions when the cumulative number of queries grows till 50,000. The gain in performance of PORQUE over PHT binary search is not significant in this context since it can no longer use direct static references. In the case of Gaussian and DBLP distributions (Fig. **??** and **??**), PHT binary search performance is very similar to PORQUE: the former is barely better with Gaussian distribution, and worse than the latter with DBLP distribution. Since it diverts the search directly to the leaf node, our approach reduces by at least 50% the number of hops in comparison with PHT linear search.

Data insertions make the trie structure grow, so we also compare the impact of data size on the search processes with respect to different distributions. Fig. **??** and **??** show the results with Uniform and DBLP distributions respectively. In the case of Uniform distribution, the cost of a PHT linear search increases faster than PORQUE. Indeed in our system the number of hops grows logarithmically with the number of leaf nodes. PHT binary search, on the other hand, exhibits a fluctuating but overall good performance. The fluctuation of the curve can be explained since the number of iterations to find a leaf node can vary depending on the size of the searched branch of the trie and the maximum key prefix size.

Query Rate: One direct consequence of the latency reduction is the increment on the number of queries processed per time unity (Query Rate). Fig. **??**, **??**, **??** show that PORQUE outperforms both PHT search methods allowing to process a higher number of queries per simulation window. PORQUE

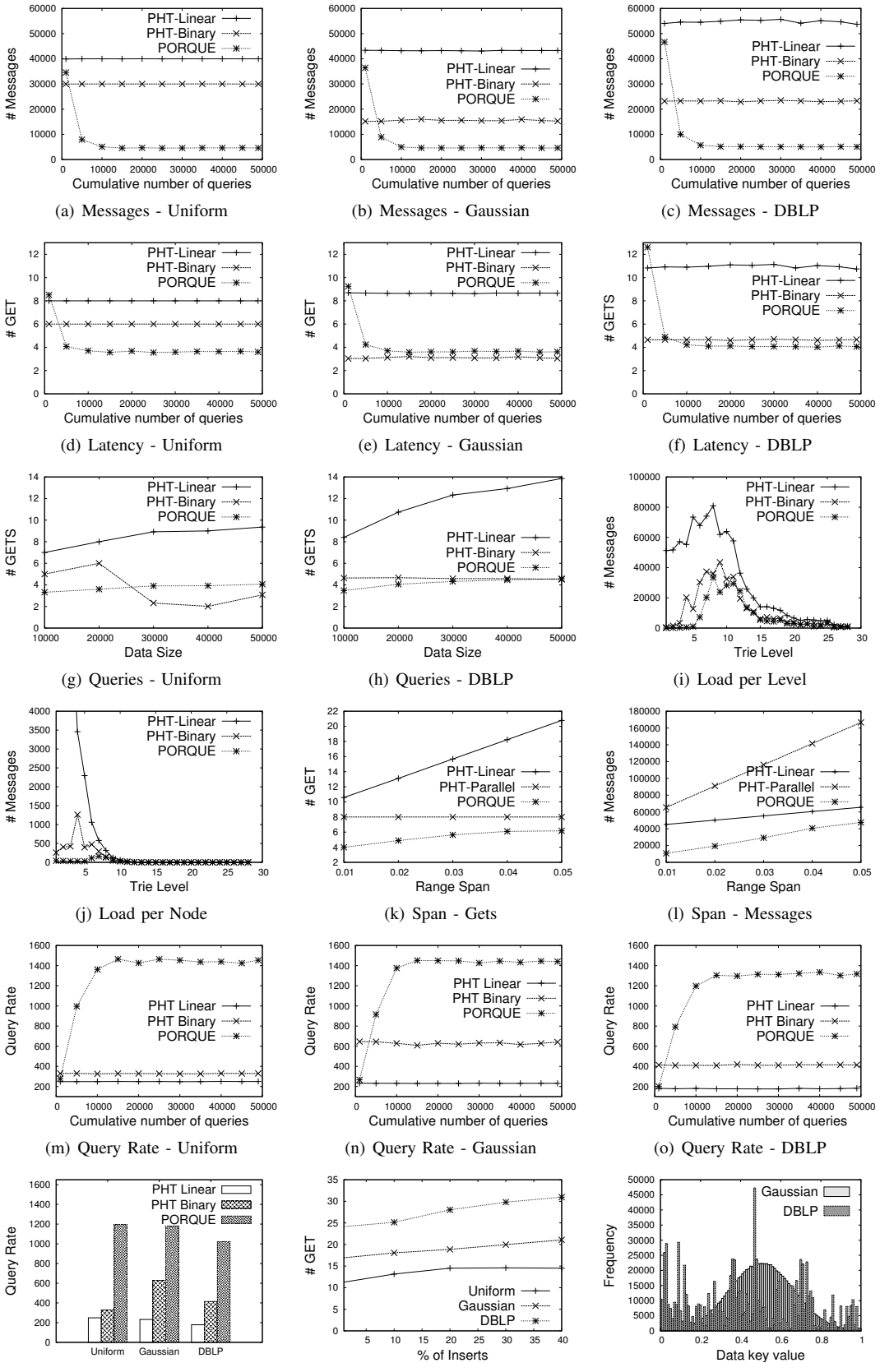


Fig. 6. Performance evaluation

can process up to 5 times more queries than PHT. Such a result is reached after performing $10,000$ when PORQUE search performance remains stable. Fig. **??** presents the average results obtained for the three data distributions.

Note that the results obtained are different from those of the previous section. Latency experiments count hops, and one hop is a DHT-lookup operation in the case of the PHT search. PORQUE, on the other hand, normally uses direct messages.

Load Balancing: One of the main advantages of the PORQUE approach is that traffic is spread over the leaf nodes of the trie. Generally, search over tree-based approaches traverse part of the trie structure to find data. The upper levels of the trie are thus highly overloaded, since queries are distributed among a small amount of nodes. This introduces bottleneck problems into the upper levels of the trie. PORQUE, however, diverts queries from the upper levels directly to the leaf nodes. Therefore, searches occur at the leaf level only and messages are distributed over a greater number of nodes. Our approach avoids bottlenecks by distributing these messages at the leaf level. In the case of PHT binary search, even if there are fewer accesses to overloaded levels, bottleneck problems still persist.

Fig. **??** shows how the load is distributed among the levels of the trie in the case of the DBLP distribution for a total of $50,000$ queries. The lowest levels of the trie have very similar loads, but PHT linear search introduces much more load on the 5 first levels. This has a significant impact on the average load per node. In Fig. **??** we can observe that the first levels incur a huge load of messages in the case of linear search, reaching $25,655$, $12,922$, and $7,148$ messages per node in the first, second and third levels respectively. PHT binary search, on the other hand, only suffers a bottleneck at level 4 (i.e., some middle level of the trie).

Range Span analysis: We have compared different strategies for gathering the results of a range query: PHT linear, PHT binary, PHT-parallel, and PORQUE searches.

Fig. **??** and **??** show respectively the number of hops and messages for PHT linear, PHT-parallel and optimized PORQUE searches for different range sizes (spans), starting from 1% up to 5% of the data space for the Uniform distribution. Linear search has the highest latency since it accesses all the nodes sequentially. Latency increases linearly with the range span. The PHT-parallel search, however, has to perform several linear searches. Since it does so concurrently, it incurs no extra time for traversing nodes that store the data: the number of hops will be equal to the longest branch that overlap the range span. Nevertheless, the traffic in terms of messages grows linearly with the range span: this can strongly degrade the performance of the system.

We can observe that the optimized search approach of PORQUE is more efficient when compared to PHT searches, reducing latency and generating lower traffic. It is worth pointing out that the slope is less steep than for the linear search and the PHT parallel search in Fig. **??** and **??** respectively. Such an improvement is the result of parallelizing the search over the leaf nodes only, and also of starting data retrieval from every node encountered inside the range as the search progresses.

Impact of Data Insertion Operation: When data insertion operations are performed in PORQUE, both the Search Table and Cache List are not updated, contrarily to range queries which are used to disseminate tables information and update their entries. However, if the insertion operation rate is similar to or greater than the query rate, the performance of PORQUE degrades. Fig. **??** shows the results of experiments performed where the percentage of the insertion operations increases with regard to range query rate. Considering $1,000$ queries, a percentage of it concerns insertion operations, and the complement are range queries. The range span is 1% of the data space. We can observe in the figure a continuous but slow degradation of the performance of the PORQUE search algorithm.

V. Conclusion

In this paper, we have presented PORQUE, a new approach for range queries on top of ring-like DHT systems. By exploiting a prefix index scheme, PORQUE maintains a second ring overlay, called Leaf Ring, composed by the nodes that store data. We have also proposed an efficient method to access this Leaf Ring using the information obtained from past range queries. Our solution efficiently reduces query latencies and message traffic.

Simulation results show that our system outperforms PHT [**?**] searches, reducing the traffic of messages by at least 50% in environments where queries significantly outnumber data insertion operations. Latency is reduced up to 50% compared with PHT. As a direct consequence of latency reduction, PORQUE outperforms PHT capability to process queries in more than 5 times. Load balancing improvements introduced in PORQUE minimize bottleneck occurrences: the load among the storage nodes is balanced, reducing, therefore, bottleneck problems in the upper levels of the indexing structure. Simulation results using different data distributions confirm that our approach has good performances, even in the presence of skewed data.

We should point out that PORQUE can be used to index multi-attribute domains by applying linearization techniques as Space Filling Curves (SFC). SFC allows to map multi-dimensional data into a single dimension. Examples of these techniques are Gray code, Z-order, Peano and Hilbert curves [**?**].