# Range Queries over Hashing

Clarissa Bruno Tuxen
Fluminense Federal University
cbtuxen@id.uff.br

Rayane de Araujo
Fluminense Federal Institute
rdearauj@syr.edu

Ioan Raicu
Illinois Institute of Technology
iraicu@cs.iit.edu

## ABSTRACT

The present work proposes the investigation of the most efficient way for implementing range queries over hashing in a Distributed Hash Table (DHT) environment. Currently, some DHTs do support such queries, however in most cases it is done in an inefficient way. One of the used approaches uses brute force by individually querying each discrete value within the range, which is not practical and could cause a lack of performance when compared to systems that could do it in an efficacious way. Oppositely, there are data structures that work well for range searches due to their internal properties, such as Red-Black trees, however they do not provide some desired DHT functionalities. When working with fast-changing attributes, and continuous values, it becomes even harder to handle the system's performance in DHT environments, that intend to uniformly distribute the keys using hashing. In this scenario, this research intends to find a suitable approach to fulfill this need as well as to reproduce it and evaluate its performance comparing with the approaches that favor both sides, the range query and the load balancing, so that those systems would achieve better performance in terms of number of lookups and time.

## Keywords

DHT; PHT; range query; hashing

## 1. INTRODUCTION AND BACKGROUND

DHTs systems do not properly support range queries themselves. A simple but non optimum manner of performing this kind of queries consists in individually querying each discrete value within a range. This type of approach is not practical and could cause performance issues that would, consequently, impede the system of working as expected. When working with fast-changing attributes, and continuous values, it becomes even harder to handle the system performance in this scenario, specially regarding DHTs that intend to uniformly distribute the keys using hashing [13]. In addition to that, the range searches would have a better response time, if each discrete value from the range is not being queried separately specially if the range is sparse, which means that there are only a few keys within the range and most of the searches would not return objects.

Zero-Hop Distributed Hash Table (ZHT) is a DHT-based data structure which has been adjusted to handle High-End Computing [11]. Based on a persistent NoSQL storage system [7], it aims at providing an available, fault tolerant, efficient structure for systems that work with a great amount of nodes. ZHT tries to merge the advantages of DHTs and efficiency in terms of latency and throughput. However, currently, it does not provide support for performing range queries, making it susceptible to face the issues presented above.

Based on the presented problems, this project has the goal of discovering an efficient approach to support range queries in a ZHT-like environment. As previously discussed, the current range query approach that some DHT systems implement is infeasible in most cases [2]. Therefore, it is the intention of this research to find an approach which avoids querying each value within the range, using a structure that provides efficient range search over hashing, since the hashing properties are desirable, aiming at being more efficient. Implementing this approach, would lead ZHT to a faster and more efficient solution regarding this task. Once the ZHT storage system is based on hashing, studying how to deal with range queries over hashing becomes an important aspect to implement such queries in this context.

A DHT data structure is a system that allows data to be distributed across nodes and easily retrieved in a scalable manner [5]. It can be seen as a large scale multi-computing hash table. More specifically, a hash table is a data structure that stores key-value pairs distributedly across the table. It does so by applying a hash function to the unique key in order to define in which entry of the table that pair will be stored. Hash tables provide $O(1)$ inserts, deletes and lookups of keys, since they all require to first hash the key and then apply the desired operation. Considering the hash function to be efficient, this structure offers uniform distribution of objects in the storing space. However, if it is intended to retrieve sequential values in a range, such structure is inefficient due to the distribution of objects. The proximity of similar items in the domain can not be explored in the hashed key space since the objects were uniformly distributed across the storing space and this is a good property to be kept once it provides load balancing.

In DHTs, instead of different entries there are different nodes and it becomes clearer why a good distribution is desired, since it provides load balancing of keys across nodes and decreases overheads, avoiding bottlenecks. Even though this data structure is widely used as a building block for several distributed applications, it is limited to only efficiently support exact match queries for the aforementioned reasons. Range queries intend to retrieve all object within a certain range of keys, making an efficient solution for this problem non-trivial in DHTs.

As for data structures that support range queries, Red-

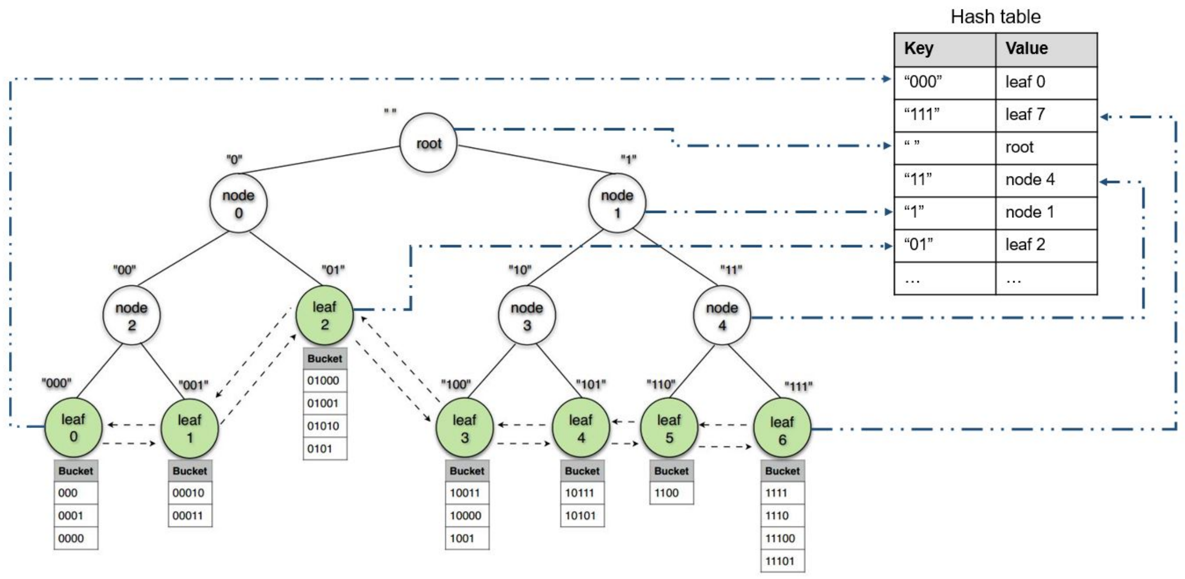

**Figure 1: PHT representation. Each PHT node is one object to be stored in the DHT. The DHT key is the PHT node lable, and the DHT value is the PHT node itself.**

Black trees provide $O(\log n)$ insert, delete, and lookup of keys [6]. The tree has five properties that may require some maintenance when inserting or removing elements but it still keeps the complexity mentioned above where $n$ is the amount of elements stored in the data structure. When comparing Red-Black trees and hash tables, these operations differ in two aspects that concern to this work. The former searches their objects sequentially starting from the leftmost leaf until the rightmost leaf is reached. So, it can retrieve all elements within a given range regardless of the sparsity of the values. It means that for a given range, if there are few objects stored that belong to this range, it does not affect the tree traversal, since the tree stores the values and during the search the nodes are visited and these values are checked against the range. The same cannot be guaranteed for hash tables. As mentioned before, hash tables do not map the proximity of similar items in the key domain to the hashed domain, so the range search needs to individually lookup each possible key in the range. This adds constraints to the key domain, that needs to allow discretization as well as the additional overhead that may change with the range density.

The second aspect that differs in both structures is the operations time complexity. While the hash table provides constant time for the basic operations (insert/delete/lookup), the tree's complexity depends on the amount of elements to be stored. This could lead to a potential scalability issue. However, for the range queries, hash tables' complexity varies according to the size of the range, while the tree's complexity varies according to the range density.

In this work, range density is based on the amount of elements contained in the range. It is related to the amount of hits and misses that occur when querying a given range. As an example, in the domain [1, 3, 5, 10, 11, 12, 13, 14, 15], the density of the range [1, 5] is three; while the density of [10, 15] is six. Thus, the second range is denser than the first one, since there were more hits.

As for distribution of objects, both structures provide it. Hash tables use their hash function to uniformly distributes the keys among the storing space, while Red-Black trees are balanced, with their properties assuring it.

Given the usability of DHTs, their mapping to hash tables, and the desired properties found in hash tables and Red-Black trees, this work intends to study a data structure that can provide efficient range querying without loosing the good performance for the basic operations.

## 2. PROPOSED SOLUTION

As mentioned in Section 1, the intent of this work is to research the possibility of efficiently implementing the lookup of range queries in a hashed structure in order to maintain the load balancing property adding the support for such queries.

The initial goal consisted of searching and analysing existing solutions, so that, based on them, improved approaches could be discussed and a better solution could be proposed. However, due to time constraints the work was focused on implementing one existing approach as well as evaluating its performance and comparing it against both structures (hash tables and Red-Black trees) that perform well for each case mentioned in Section 1, with the nature of the work being a real system development.

Since DHT can be seen as a multi-computing hash table, as mentioned in the Section 1, a single-node implementation of a system using PHT over a hash table was evaluated in order to simulate such approach. Once PHT is not an open source project, the approach was implemented following the properties described in [13]. It was implemented using Java. The HashMap and TreeMap Java classes were used to represent the hash table and the Red-Black tree, respectively. The Pair class in the $javafx.util.Pair$ is the only dependency outside of the $java.util$ package.

Among the approaches found, PHT [13] was selected, because it could be suitable for integrating with the ZHT structure. This approach suggests the use of an overlay network

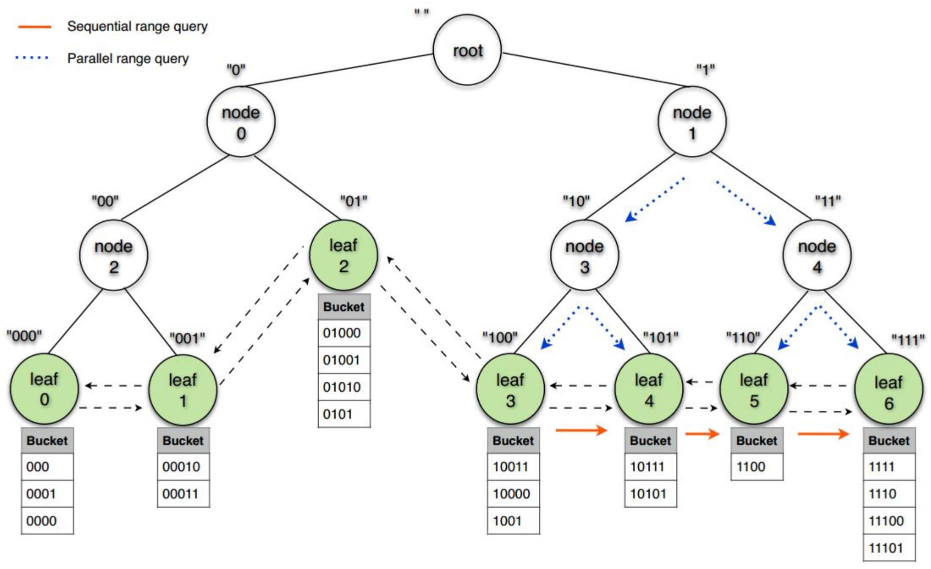

**Figure 2: PHT range query algorithms representation. The sequential algorithm finds with the prefix of the lower bound and traverses the linked list of leaves until the leaf containing the upper range is found. The parallel algorithm finds the smallest prefix that matches both boundaries in the range and from this node it looks for leaves matching the range in parallel. When in the leaves, the buckets are checked and the keys within the range have their values retrieved.**

over the DHT layer. A good aspect of it is the fact that it can be applied over any DHT once it does not require any alteration in the DHT hashing function. The PHT approach suggests the usage of a binary trie that maintains the distance property between its elements by using prefixes. Because of that, the number of lookups is minimized and the range query process is simplified and optimized. A PHT overview can be found in the section 2.1.

Alongside with PHT, the naive brute force approach was implemented using a hash table and providing range queries by looking up each discrete key value within the given range. As discussed earlier, even though this approach is not efficient, it is the most intuitive way of performing such queries in a hashed data structure.

Finally, the last approach considered was a Red-Black tree, which is a balanced binary tree that stores its elements in a sorted manner. It was implemented in order to provide a comparison with an structure that can efficiently retrieve range values.

A more detailed description of PHT is presented in the following section as well as an understanding on how the range queries are performed.

## 2.1 Prefix Hash Tree

PHT is a resilient and efficient trie-based distributed data structure that implements more complex queries over a DHT [13]. It is an overlay network that constructs a trie-based structure over a DHT and takes advantage of its lookup interface. Once PHT does not require any changes in the DHT layer, it can be used with any DHT.

PHT is essentially a binary trie build over the DHT. Every PHT node has a label (its prefix) and nodes can be either internal nodes or leaves as in regular trees. Only the leaf nodes store the key-value objects in an internal data structure, the bucket. The buckets have a maximum capacity, the block size. An additional characteristic of leaves is that they are part of a doubly linked list.

The PHT properties try to maintain the trie as concise as possible. Nodes are created as leaves. During an insert, they may become internal nodes if their bucket's capacity the block size) is exceeded, so they need to create children to distribute their key-value pairs among them. This is the split property. Similarly, during a delete, a merge operation may be required. The five properties are described in [13].

The way that PHT works over a DHT is by considering the PHT-nodes the objects to be stored in the DHT. The key is the PHT-node label, and the value is the PHT-node itself. So, it is important to highlight that the PHT-nodes are different from the DHT-nodes. While the former holds the actual information, the latter stores the PHT-nodes. Figure 1 illustrates it.

In this work, the PHT approach was implemented following its properties. In order to do so, the main operations of PHT (lookup, insert, split, delete, merge, and range query) were reproduced. A description of each one of them is provided as follows.

### 2.1.1 *PHT basic operations*

The basic operations that need to be supported include the ones that provide the basic functionalities (insert, delete, lookup) and the ones that maintain the PHT properties (merge and split).

The lookup operation intends to retrieve the value of a given key. It can be seen as containing two steps: finding the PHT node (PHT-lookup) and retrieving it from the DHT (DHT-lookup); and retrieving the value of the desired key. The first step consists of tring different prefixes and using them as the key for the DHT-lookup. It can be performed either linearly or as a binary search. If the DHT-lookup returns a PHT leaf node, this step is finished; otherwise,

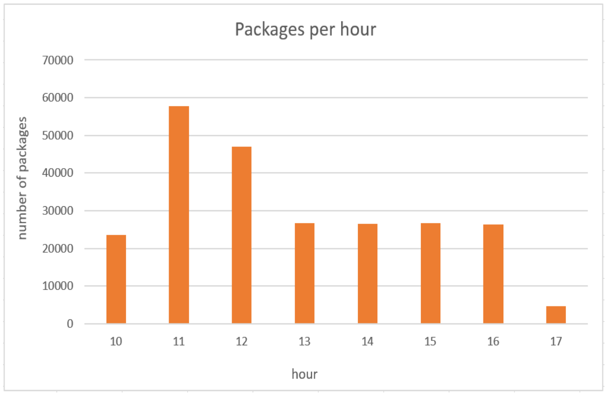

**Figure 3: The network trace hourly package distribution.**

the next prefix is decided based on the algorithm (linear or binary seacrh) and used as the DHT key. In the second step, the PHT-node's bucket is traversed in order to retrieve the value represented by the desired key. The present work only reproduced the linear lookup due to time constraints and the fact that the focus of this research is the range query itself.

For the insert operation, a DHT-lookup will be done to retrieve the PHT-node in the same way as the first step of the lookup operation, retrieving a leaf node. If the bucket of this node is already full when inserting a new value, a split operation can become necessary.

A split operation is only performed when the PHT-node's bucket is full when a new object is inserted. In order to keep the PHT's properties, this requires the overloaded PHT-node to create two children (because of one of the properties) and split its objects among them. It is interesting to notice that this may infer further splits, since it is possible for all objects to go to the same child, making it overloaded and requiring a split to be performed.

The delete process is also straight forward. A DHT-lookup will be done in order to find the PHT leaf node that contains the given value, once again in the same process as the first step of the lookup and the insert. After the PHT-node is retrieved, the object with the given key is removed from the bucket, therefore it is removed from the entire system. In this process, the inverse problem of the insert can happen. It may be necessary to merge nodes into an ancestor.

The merge operation happens in order to maintain another property that states that there is a minimum amount of keys to be stored in each internal node's sub-tree. This number is related to the block size (bucket size), more precisely $blocksize + 1$. This operation only ensures that each internal node has no less than two children and it also helps the tree to stay concise.

#### 2.1.2 PHT range operations

Regarding the range query process, [13] also proposes two ways of doing it. The sequential algorithm performs DHT-lookups for the PHT-node that is a leaf and is labeled with the lower (or upper) bound prefix. Once such node is found, all keys in the bucket are checked against the range and those that belong within the range have their values retrieved. While the upper (or lower) bound is not reached, the leaves are traversed through the linked list and the check of the keys in the buckets is performed. The entire operation requires one PHT lookup operation at the beginning and it adds the cost of traversing the amount of leaves that contain keys within the range.

In the parallel range query, DHT-lookups will also be performed in order to retrieve the node whose label is the smallest prefix that covers the whole range. If it returns an internal node, the search is forwarded to the children that overlap the range until leaves are reached. Once this happens, the buckets are checked and the values whose keys belong to the range are retrieved. If it is a leaf, only the final step in necessary.

## 3. EVALUATION

This section presents a single node evaluation of the PHT data structure. A comparison between this work's implementation and the simulation in [13] is shown. Also, in order to evaluate the PHT's time performance, a comparison between PHT, hash tables, and Red-Black trees is performed. Two different datasets were used in the experiments, synthetic and network trace.

The following sections detail the environment setup and the datasets.

### 3.1 Testbeds and Benchmark details

As mentioned in Section 2, all three data structures were implemented in Java. The hash table and Red-Black tree used Java's classes $HashMap$ and $TreeMap$, thus only an interface was implemented using these classes inherent operations.

The experiments were run in Ubuntu 16.04 LTS OS with AMD Phenom II X6 1100T processor and 16 GB of RAM. The implementations are single threaded and single node, using the fact that the objects can be kept in memory.

### 3.2 Synthetic dataset

For this dataset, the experiment setup is composed of the generation of $2^{16}$ 30-bit keys coming from an uniform distribution in a key space of $2^{30}$. The PHT block size $B$ is set to 20. A java $HashMap$ is used as the hash table layer in which the PHT nodes will be stored. The focus of this experiment is validating this work's implementation against the results presented in [13].

The metric used for evaluating this dataset is based on the optimum amount of traversed leaves [13]. The optimum value ($O$) can be obtained from the size of the result set ($R$) and the block size ($B$). It can be described as $O = \lceil \frac{R}{B} \rceil$. This represents the ideal case, in which the keys are well distributed across leaf nodes and only the minimal number of leaves needs to be traversed.

### 3.3 Network trace dataset

The network trace dataset was obtained from the "The industrial cyber security conference" (4SICS 2015) [1] and it has the network trace of the ICS lab. The trace contains information of approximately 7 hours, from approximately 10 am until around 5 pm. There is a total of 239,267 entries. The distribution of the entries per hour is shown in Figure 3.

The experiments run with this dataset intended to investigate the lookup performance difference between PHT and the hash table, and the time performance difference between

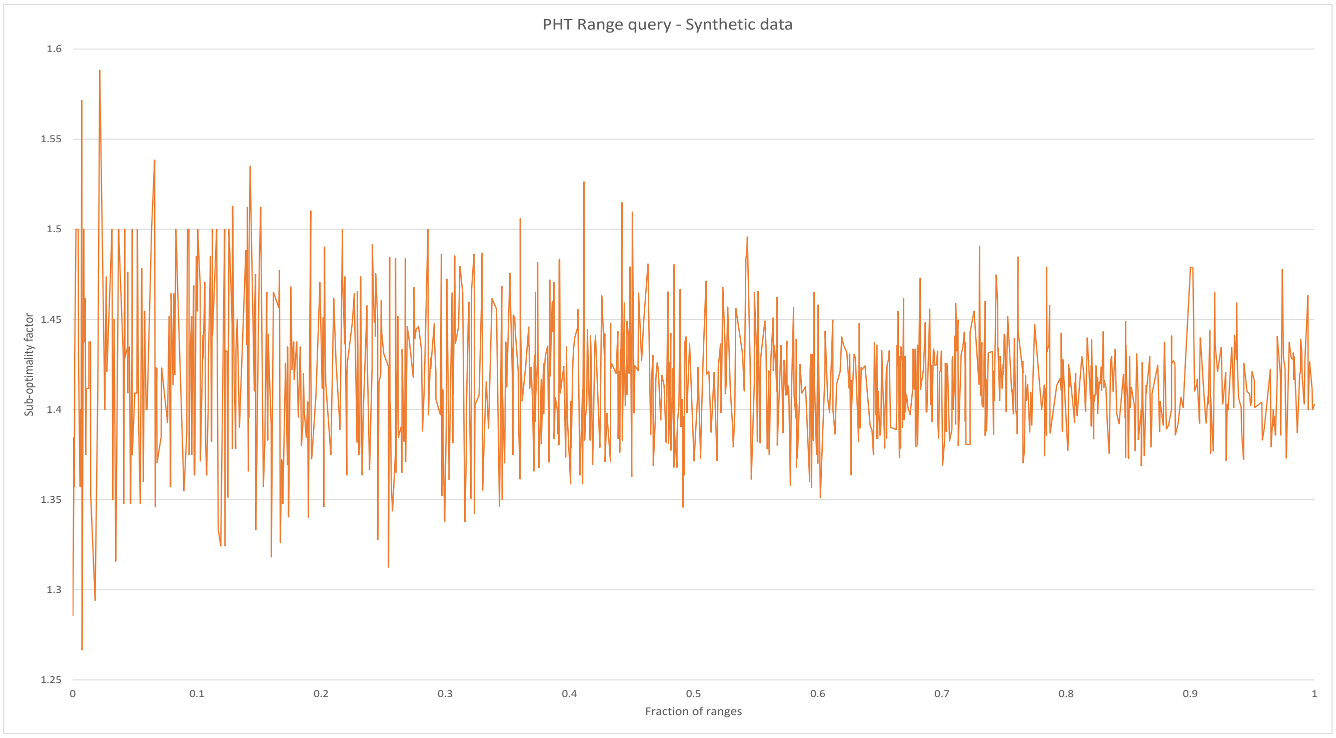

**Figure 4: The sub-optimal value of the leaves traversed given the range size fraction for the synthetic datset. The average sub-optimal factor is 1.417.**

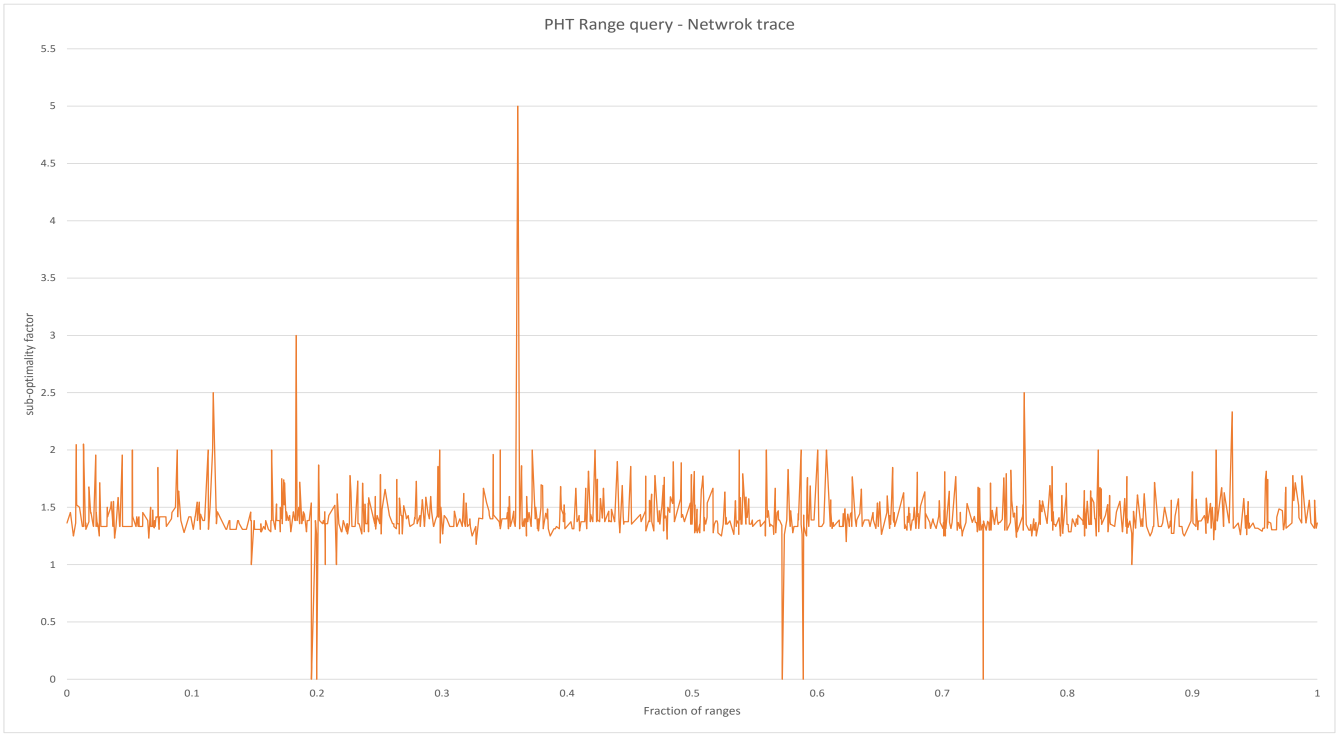

**Figure 5: The sub-optimal value of the leaves traversed given the range size fraction for the network trace dataset. The average sub-optimal factor is 1.427.**

the three data structures. In the latter, an average of three runs was considered.

In the time performance analysis, three different workloads were considered in order to evaluate the impact of the range density. The workloads do not impact the lookup performance, since it is mostly based on the range size rather than the result set size.

## 3.4 Workload definition

Using different workloads is intended in order to explore different range densities. The range density is based on the result set size. It is related to the amount of hits and misses that happen when a range query is issued. More hits indicates a denser range, while more misses indicates a sparser range.

The workloads are defined by the amount of objects inserted in the data structure. This intends to induce different range densities for each workload. The three workloads had their objects selected from the network trace dataset with a random choice of entries coming from an uniform distribution. They are defined as follows.

- The first workload inserts 100% of the objects, leading to denser ranges;
- The second, uses 60% of the data, with intermediary density;
- The third, considers 20% of the objects, leading to sparser ranges.

A distribution of the result set size per range size for each workload is observed in Figure 7. As the points have a higher result set size, the ranges are denser since more hits happened. Oppositely, when the result set size is smaller, the range is sparser which indicates that less hits (more misses) will happen when querying a given range. As a general trend, the 100% workload tends to be denser given its higher values, while the 20% workload is sparser since it has generally result sets with fewer values. The 60% workload presents an intermediary density having its values between the two more extremes workloads. This corroborates the assumption that inserting different amount of elements would lead to different range densities.

## 3.5 Range Query Evaluation

All range queries experiments had a 1000 randomly generated queries. For the synthetic data, the ranges varied from $2^{22}$ to $2^{26}$. For the network trace data, the ranges varied from 30 seconds to 1 minute. All random values come from an uniform distribution.

The first experiment used the synthetic data as explained in Section 3.2. Figure 4 shows the amount of leaves that were actually traversed normalized by the optimum value (minimum) in PHT. It shows the sub-optimal value for a given range size. The x-axis represents the range sizes normalized into a (0,1) scale. Even though there is some variation in the values, the average (1.417) is comparable to the average of the values obtained by the simulation in [13] (about 1.4). The chart can lead to the intuition that the split operation interferes in the actual amount of leaves traversed based on the spikes shown.

When the same analysis is performed with the network trace dataset in PHT, Figure 5 indicates that, in average, the sub-optimal factor is roughly the same, being 1.427 for this dataset. The spikes are attributed to the fact that some queries traverse leaves that have empty buckets. This happens because the split property of PHT is being followed. So, after a split operation happens, it is possible that all keys are assigned to the same child leaving the other child with an empty bucket and requiring a new split. If the same situation happens, many leaves with empty buckets will be created, thus increasing the amount of leaves that need to be traversed for a range that starts before such leaves and ends after.

Figure 6 shows the amount of DHT-lookups under the denser range (100% workload). The amount of DHT-lookups performed by the hash table is considerably worse than PHT's. The density of the range does not affect the number of lookups of the hash table, however it can be increased in the PHT. Still, PHT performs better than the hash table by a large margin.

The time performance analysis uses the three aforementioned workloads. Given the range density difference with different workloads, the time performance analysis presented in Figure 8 compares the total time of range query operations between the three data structures. It can be observed that the hash table performance is considerably worse than the others even though it is consistent across range densities. This can be attributed to the fact that the hash table's range query complexity is based on the range size rather than the result set size, which influences the density.

As for PHT, there is almost no variation with marginally better results for the intermediary workload. A factor that could influence these results is the split property. With sparser ranges, the amount of leaves with empty buckets could be higher and add some overhead to the operation. However, overall, PHT clearly outperforms the other two data structures.

The Red-Black tree has results affected by the density, with the best time for the sparser workload. This is aligned to the structure's properties, since the sparser workload results in smaller result sets. The Red-Black tree range query complexity was said in Section 1 to depend on the result set size which can be confirmed in this Figure.

As an overall comparison, given the intermediary workload, Figure 9 presents the times for the fastest query, the slowest query, the average, and the total time of the 1000 range query operations. It is clear that PHT has the best performance. The variation between the fastest query and the slowest query for the hash table is small with both presenting a large value. As for PHT and the Red-Black tree, both have low time for the fastest query, however they differ in the slowest query (2 milliseconds for PHT and 47 milliseconds for Red-Black tree). The tree has a larger variation which indicates that PHT is in overall faster as it can be seen in the total time. It is important to notice that the average PHT time is not negative, since the y-axis is in a log scale.

Figure 10 shows the time performance with three range sizes variations for the intermediary workload (60%). The range sizes vary at most 30 seconds, 15 seconds, and 7.5 seconds. It is demonstrated that PHT and the Red-Black tree have consistent times with PHT clearly outperforming both data structures. The hash table presents some improvement as the range size decreases which supports the initial assumption of the hash table time complexity being based on

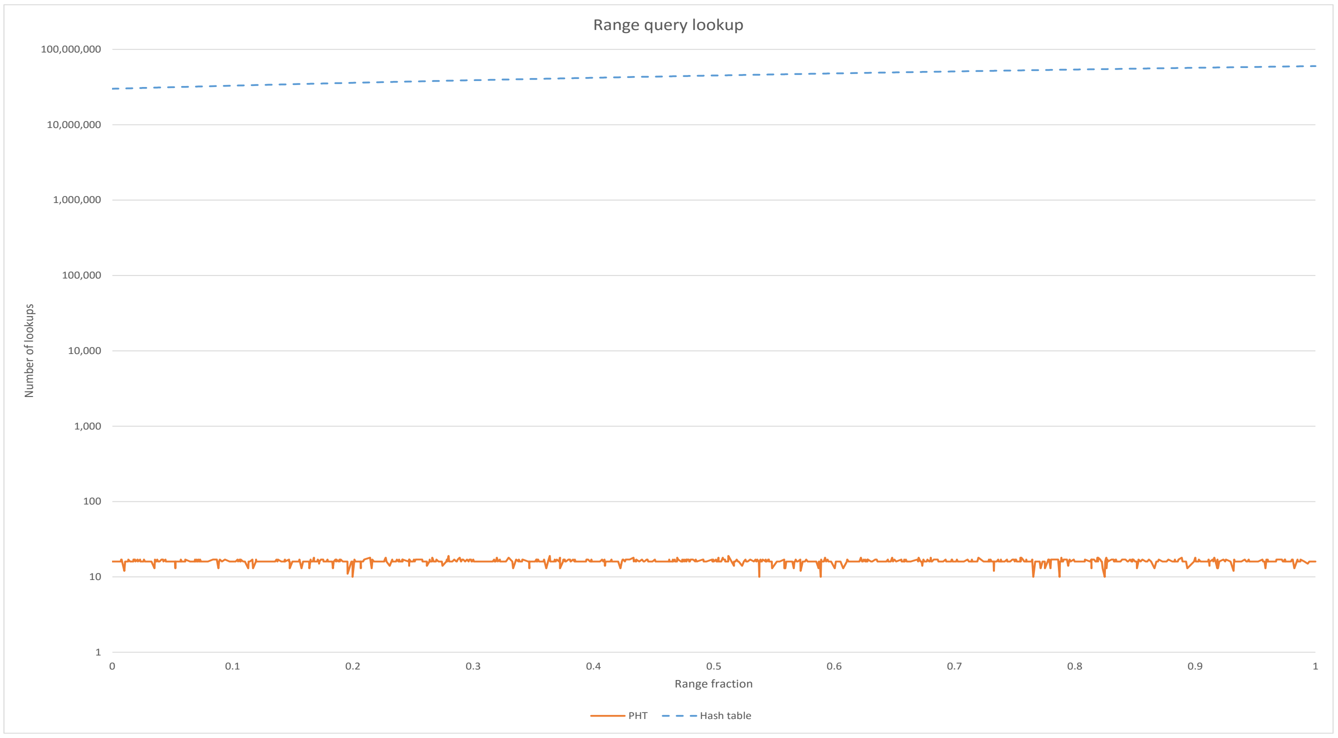

**Figure 6: Comparison between the hash table and PHT regarding the number of lookups performed when a range query is issued. The x-axis represents the range sizes normalized in a $(0, 1)$ scale.PHT has an average of $16$ lookups while the hash table's is $44, 841, 724$ lookups.**

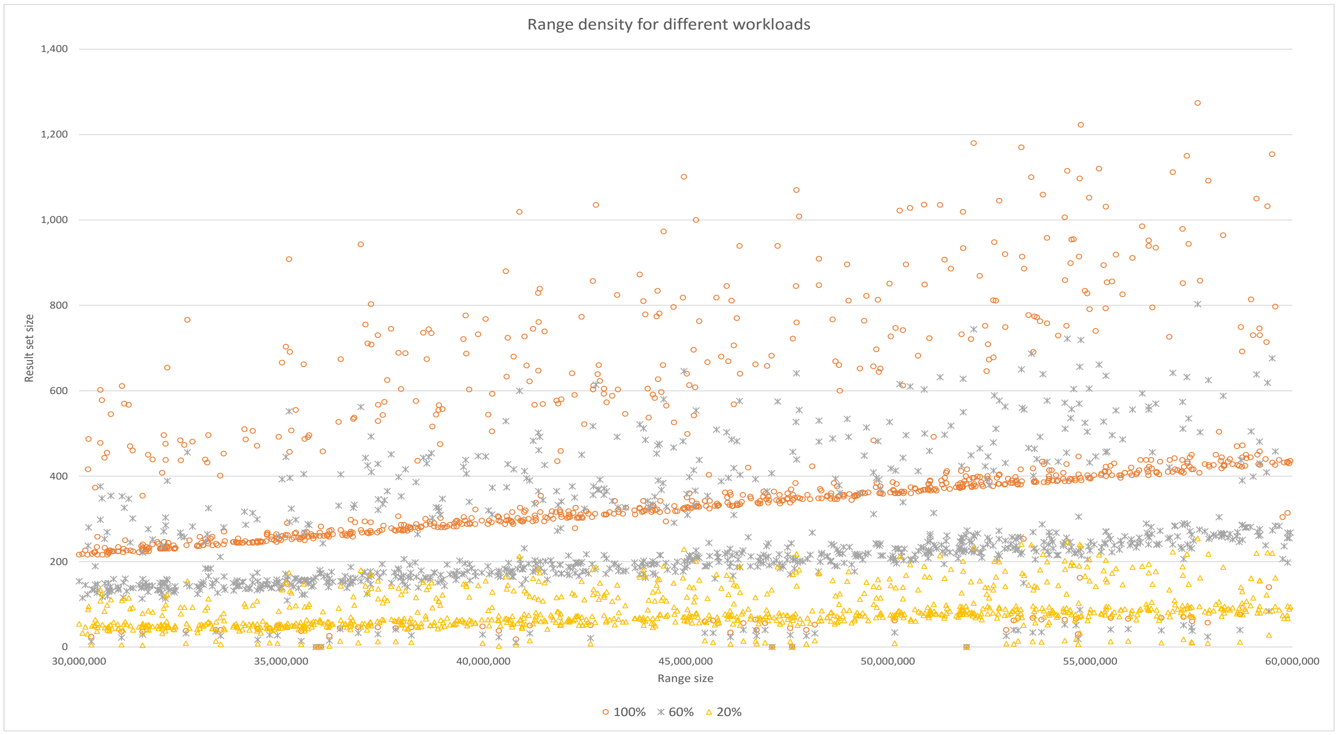

**Figure 7: Result set size distribution by range size. Higher values indicate denser ranges, since there are more hits within the given range. The opposite can be said about the lower values, where less objects will be found, making the range sparser. Based on this, the $100\%$ has denser ranges, the $60\%$ has intermediary ones, and the $20\%$ has sparser ranges.**

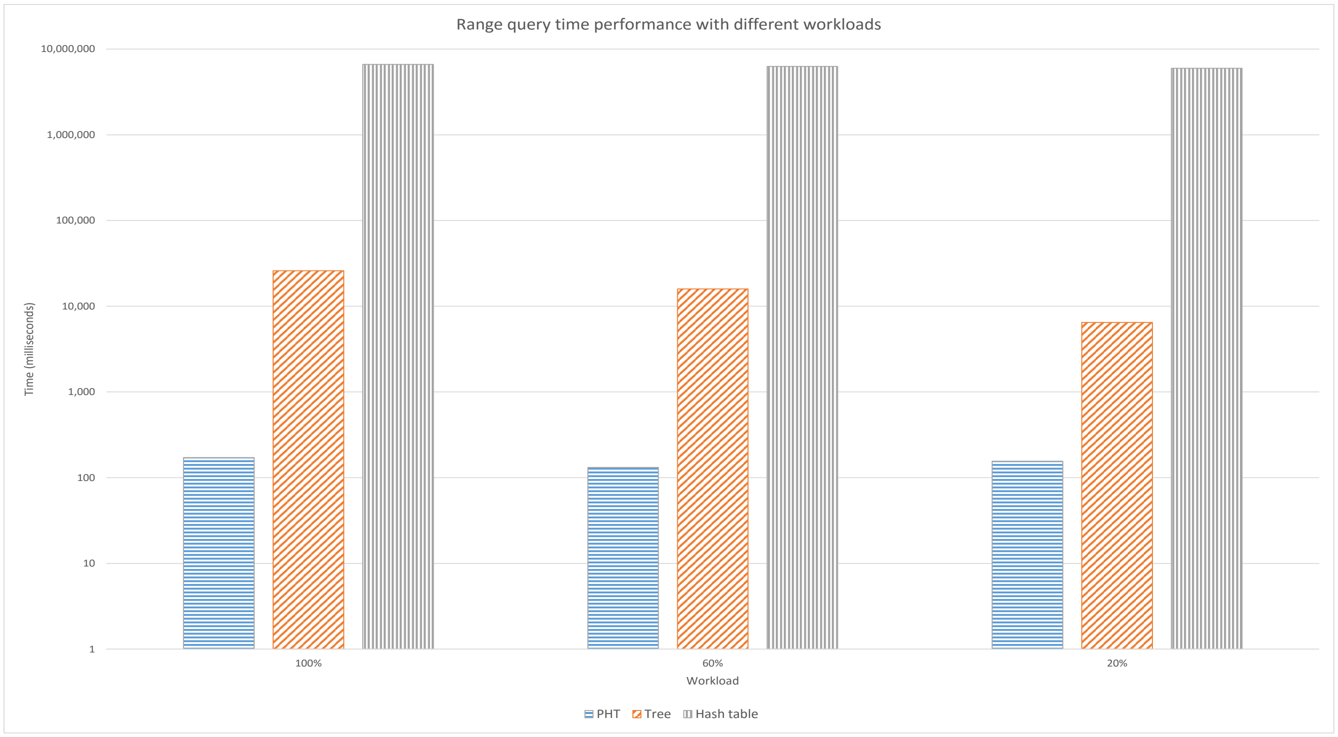

**Figure 8: Total time performance of PHT, hash table, and Red-Black tree. PHT outperforms both data structures showing a consistent time across workloads, marginally better in the intermediary workload. The hash table presents the worse results, however consistent, since its range query time complexity depends on the range size rather than result set size. Oppositely, the Red-Black tree has the most affect values which also consists with its complexity being dependent on the result set size.**

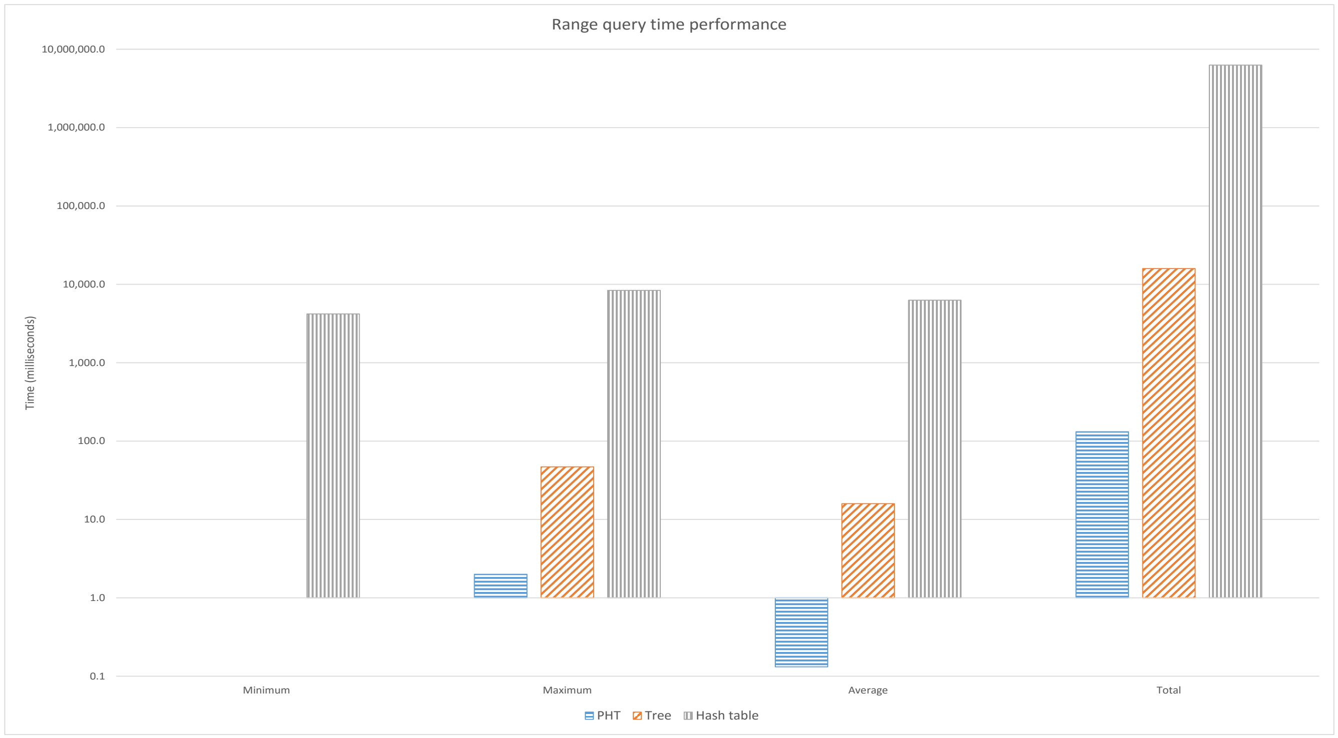

**Figure 9: Time performance for PHT, hash table, and Red-Black tree with the intermediary workload. PHT outperforms the other data structures in all time measurements.**

the range size. This Figure alongside Figure 8 display the different time complexity dependencies for the Red-Black tree and the hash table. The former shows the Red-Black tree dependency on the range density, while the latter illustrates the hash table dependency on the range size.

## 4. RELATED WORK

The presented issue was not extensively discussed previously, therefore there are not many approaches proposed for it. During the investigation, a few were found and they are presented below.

In [2], the implementation of range queries in a Peer-to-Peer (P2P) CAN-based network is discussed. In addition to that, the authors are concerned about the problems that some DHTs can present when performing such queries by individually querying each discrete value within the range. To solve those problems, this paper presents the usage of a subset of servers that will act as nodes in this network. Those servers will act as Interval Keepers (IKs) and will store the pairs [attribute-value, resource-id] and will be responsible for a sub-interval of values within the range. They use Space Filling Curves and the Hilbert Function [3] to map each sub-interval within the IK and its corresponding zone in the dimensional space. In this way, the zone will be divided into sub-zones of equal spaces. The range query will be routed to the middle point and this node will recursively propagate the query to its neighbors. This process is called Flooding and they present three strategies to implement it: Brute Force, Controlled Flooding and Directed Controlled Flooding. Both PHTs and this approach split the domain, however PHT has an overlay layer over the DHT, which simplifies the approach and makes it more versatile since it does not inflict changes in the DHT layer.

In [4], Skip graph is presented. It is a data structure based on skip lists that is meant to handle complex queries. It is tolerant to node failure and preserves the order of the keys. [12] combines Skip Graphs with a traditional trie. The basic idea of this approach is to take the advantages of Skip Graphs for efficient routing, while using trie to preserve locality. Another application of Skip Graphs can be found in [9] where the authors propose a structure that based on a skip tree, supports aggregation queries and outperforms range queries and exact-match queries in skip graphs. However, even though Skip Graphs seems to be a good approach for complex queries it does not provide load balancing. In addition to that, this work has the intention of being aggregated to ZHT what makes the Skip Graph approach not possible once it is a different data structure itself and it does not provide load balancing, unlike PHT.

B-Trees can also support complex queries. In this structure, all leaf blocks are at the same tree level and the data is only added to the leaf nodes. In addition to that, B-trees are not sensible to clustered data. [10] proposes a P2P overlay network based on a balanced tree structure. The authors claim it to be fault tolerant, load balanced and efficient regarding costs for update operations. Although B-trees can be used to perform range queries, for the goal of this work it does not seem to be a good option. It requires an initial tree traversal to access any block, which can increase latency when performing range queries. In addition, it does not support concurrency for some operations and similarly to [4], it proposes the usage of a new data structure, the B-tree itself. Unlike PHT, this approach does not propose a tree structure over hashing which makes a further integration with ZHT not practical.

Finally, in [8] the authors also use tries in order to improve the range query process. They propose the usage of an "in-network indexing" in an overlay network to make it efficient for high-level predicates. Instead of using uniform hashing, this work proposes using an order preserving hash function to optimize the process. In this approach, the P-Grid DHT is used in order to provide the aforementioned properties. While this structure maintains the structural order of the data, it may have skewed data, compromising the load-balancing property obtained when an uniform distribution of data is used. They claim to offer a logarithmic search regarding the number of messages for exact match and range queries. However, such approach does not seem highly available if the workload is excessive, due to the imbalanced distribution of data and potential congestion.

## 5. CONCLUSION

During the time of this research, an investigation of the most efficient way of performing range queries in a DHT environment was done. For this purpose, a better understanding of DHT as well as an improvement of general research and team work skills was provided to the students involved. A basic study about ZHT and NoVoHT was also done by the group members with the goal of better understanding its functionality and storage system so that the range query approach chosen would match ZHT's need.

The evaluation presented intended to compare PHT's performance against data structures that perform it well for the two different query cases: the exact match (lookup), and the range query. Hash tables have $O(1)$ time complexity for the basic operations (insert, delete, and lookup) and the analysis predicts range query time complexity to be dependent on the range size rather than the result set size. So, it is expected to perform badly for range queries. Red-Black trees have $O(\log n)$ time complexity for the basic operations with $n$ as the amount of elements in the data structure. Regarding range queries, their complexity is expected to be dependent on the result set size (range density) rather than the range size. So, their performance is predicted to vary as the amount of elements retrieved varies regardless of the range size.

It was also intended to evaluate the data structure's performance regarding range density, since it was possible that they could have a comparable performance in a denser range. Hash tables have to query the entire range while Red-Black trees' performance is dependant on the result set size (range density). So, experiments with denser ranges could provide similar results for these structures. However, this was not observed, since the tree outperformed the hash table and PHT outperformed both.

Based on this work's evaluation in a single node, PHT has shown the best performance regardless of the range density or size, outperforming both data structures by a large margin. It has proven to be effective and consistent as the range density varies, unlike the other two data structures. The hash table has similar results across range densities, while the Red-Black tree shows an improvement as the range becomes sparser, but it is still worse than PHT.

As mentioned before, this work had the goal of investigating the most efficient way for performing range queries in a DHT environment. However the presented results were ob-

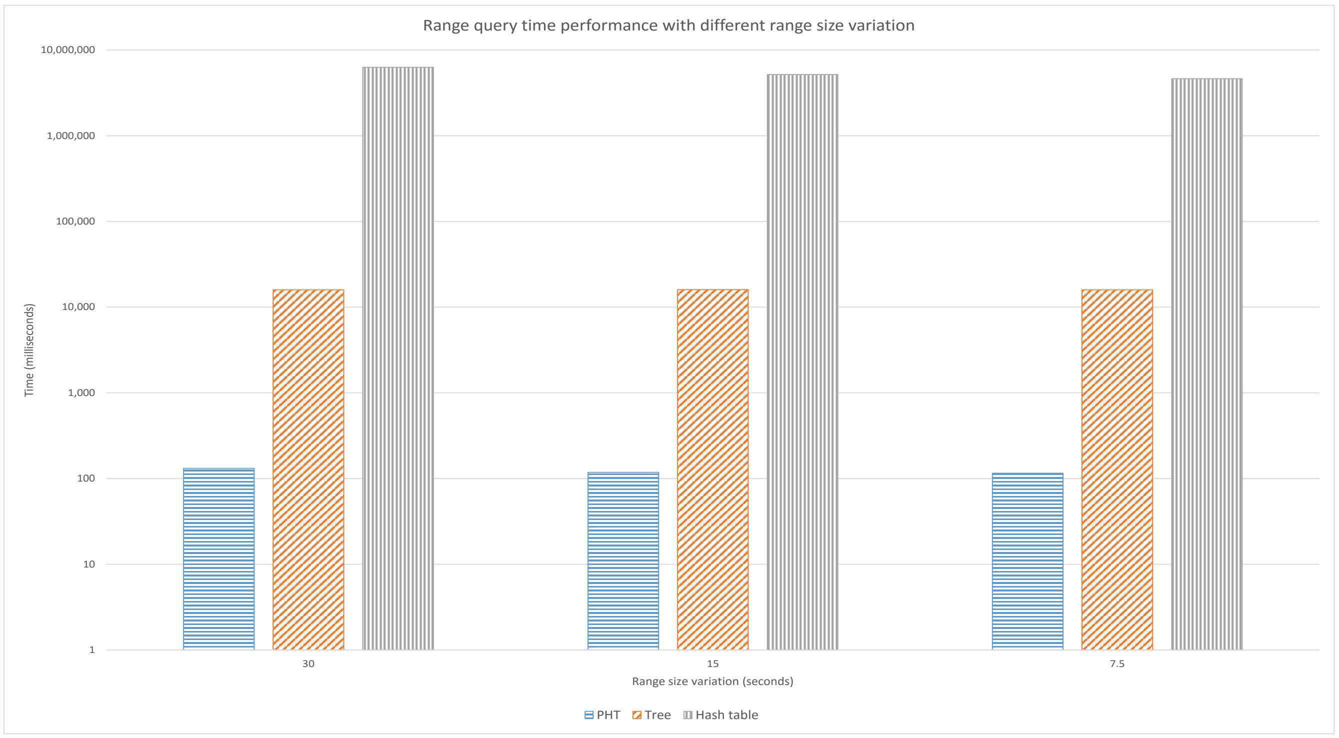

**Figure 10: Time performance for the intermediary workload (60%) with three different range sizes variations. It is shown that the Red-Black tree time is consistent while the hash table is sensitive to the range size since as the range size decreases, the hash table performance improves. PHT presents the best performance in the three cases while showing consistency regardless of the range size.**

tained from a single node implementation. As future work, this data structure could be extended to a distributed setting in which the evaluation and comparison against the results presented in this work could be performed. Once the system becomes distributed, the integration with ZHT seems worth investigating. It would be interesting to evaluate if ZHT would be able to support range queries with good performance using PHT. Another possible investigation is regarding the range query algorithm used in PHT. [13] proposes a sequential and a parallel search algorithms. In this work, only the sequential algorithm was used in the experiments.

As another possible improvement that could be investigated, the lookup algorithm could have the data keys stored in the DHT. This would decrease the amount of DHT-lookups, since a single DHT-lookup for the data key would be required and it would point to the PHT-node that holds such key. Even though the lookup time is decreased, the maintenance time will potentially increase once every time a split or merge operation happens, some keys would be moved to different PHT nodes and the hash table value for these keys would need to be updated. An alternative can be postponing the DHT updates until the lookup for the key is performed and the information stored in the DHT is outdated. When this happens, the regular PHT lookup can be performed and once the key is found its value (held by the PHT-node) can be updated in the hash table. It is expected that even with the selective update, the lookup time will decrease in average.

## 6. REFERENCES

[1] 4sics - 2016 - home.

[2] A. Andrzejak and Zhichen Xu. Scalable, efficient range queries for grid information services. pages 33–40. IEEE Comput. Soc.

[3] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space filling curves and their use in the design of geometric data structures. In R. Baeza-Yates, E. Goles, and P. V. Poblete, editors, *LATIN '95: Theoretical Informatics*, volume 911, pages 36–48. Springer Berlin Heidelberg.

[4] J. Aspnes and G. Shah. Skip graphs. 3(4):37–es.

[5] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in p2p systems. 46(2):43.

[6] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. 1(4):290–306.

[7] K. Brandstatter, T. Li, X. Zhou, and I. Raicu. NoVoHT: a lightweight dynamic persistent NoSQL key/value store.

[8] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range queries in trie-structured overlays. pages 57–66. IEEE.

[9] A. González-Beltrán, P. Milligan, and P. Sage. Range queries over skip tree graphs. 31(2):358–374.

[10] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 661–672.

VLDB Endowment.

[11] T. Li, R. Verma, X. Duan, H. Jin, and I. Raicu. Exploring distributed hash tables in HighEnd computing. 39(3):128.

[12] L. Meifang, Z. Hongkai, S. Derong, N. Tiezheng, K. Yue, and Y. Ge. Pampoo: An efficient skip-trie based query processing framework for p2p systems. In M. Xu, Y. Zhan, J. Cao, and Y. Liu, editors, *Advanced Parallel Processing Technologies*, volume 4847, pages 190–198. Springer Berlin Heidelberg.

[13] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. In *Proceedings of the 23rd ACM symposium on principles of distributed computing*, volume 37.