INTERNATIONAL JOURNAL OF INNOVATIVE TECHNOLOGY AND CREATIVE ENGINEERING (ISSN:2045-8711)
VOL.6 NO.8 AUGUST 2016, IMPACT FACTOR:0.61

Improved Index Based Over Distributed Hash Table for Peer To Peer Systems

D.Kavitha
Assistant Professor, Department of Computer Applications,
Nandha Arts and Science College,
Erode, Tamil Nadu, India.
Email:rkjaidanya46@gmail.com

Abstract – In this fastest growing trend in the Peer-to-Peer (P2P) systems, most of the current internet applications scalability becomes key issue in the performance of query exchange between peer nodes. Hashing forms the basic elements of peer networks with more literary works presented in Distributed Hash Table (DHT). The scalability of peer networks demands more complex queries to be handled. The complexity of the queries arise in terms of range, nearest neighbor, sequential and hierarchical patterns. Here consider DHT for peer networks, where uniformity of hashing is accomplished for data localization, it is unable to handle complex queries. In addition to query inefficiency, the DHT also have the history of poor maintenance. This proposed work presented an improved index version for LIGHT, which adapts a B+ tree indexing. B+ tree index deleted unwanted and unused leaf nodes to make the hash structure more effective for query exchange between peer nodes. A pseudo column is introduced by B+ tree model to increase the throughput of the peer node query exchanges. Experimentation is carried out to testify the performance of LIGHT and Improved LIGHT with B+ tree indexing in terms of data exchange overheads, throughput of query exchange, memory requirements and time to prune the query. The results indicated that proposed improved LIGHT scheme had nearly 75% improvement of throughput and 40% of overhead is minimized when compared to that of existing LIGHT. The memory requirement and pruning time are significantly reduced for the proposed improved LIGHT scheme compared to the existing.

Keywords- DHT, P2P, PHT, RST.

1. INTRODUCTION

Distributed Hash Table is a class of a decentralized distributed system that provides a lookup service similar to a hash table pairs are stored in a DHT and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participant causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

DHTs form an infrastructure that can be used to build more complex services, such as distributed file systems, peer-to-peer file sharing and content distribution systems, cooperative web caching, multicast, any cast, domain name services, and instant messaging. Notable distributed networks that use DHTs include Bit Torrent's distributed tracker, the Kad network, the Storm botnet, YaCy and the Coral Content Distribution Network.

2. RELATED WORKS

P2P data indexing has recently attracted a great deal of research attention. Existing schemes can be classified into two categories, over-DHT indexing paradigm and over lay dependent indexing paradigm. While over-DHT indexing schemes treat data indexing as an independent problem free from the underlying P2P substrates, overlay-dependent indexing schemes are intended to closely couple indexes with the overlay substrates. In the design of DHT overlays, the primary concern is topological scalability in terms of two aspects: the diameter, which determines the bound of the hops of a lookup operation, and the degree, which determines the size of the routing table.

Many proposed DHT overlays, including Chord [19] are based on the Plaxton Mesh which achieves a diameter of (β-1)logβ N and a degree of logβ N. Here, β indicates the base of the DHT identifier space.

The classical DHT, CAN [21], leverages the d-torus topology, which bears a diameter of 1/ 2 dN1/2 d and a degree of 2d.

In the over-DHT indexing paradigm, the DHT and data are loosely coupled by the keys generated from data records. Thus, a critical issue in the design of an over-DHT index is how to generate the DHT keys regarding data locality. In this category, the PHT [22] is a representative solution for range queries, and is the most relevant scheme to our proposed LIGHT. Thus, below project first introduce PHT in detail. After that, project presents other indexing schemes that support various queries for database and information retrieval applications.

PHT, as the first over-DHT index proposal, PHT supports indexing bounded one-dimensional data. Essentially, PHT partitions the indexing space with a tree structure, where all data records are stored on leaf nodes. The tree structure is materialized over the DHT in a straight forward way all tree nodes, including internal nodes and leaf nodes, are mapped into the DHT by directly hashing their labels of binary representation.

On the contrary, all data records in the children nodes would be relocated according to the parent's label during a merging process. The range query processing in PHT involves forwarding the query from the root to all candidate leaf nodes in parallel. To facilitate traversing candidate leaf nodes, PHT further maintains links between neighboring leaves, which however incur extra index maintenance overhead.

RandPeer [1] applied PHT to a specific scenario-indexing membership data for QoS sensitive P2P applications. Other indexing schemes for range/k-NN queries, several other studies have also investigated data indexing for range and k-NN queries, with their major focus being how to improve query latency by data replication.

The tree structure is static and globally known, the internal nodes can be located by a single DHT-lookup, rendering the range query solved in O (1) time. However, due to data replication in all ancestors, some high-level tree nodes could easily be overloaded. To address this issue, RST [13] employs a novel data structure called Load Balancing Matrix (LBM), which organizes overloaded tree nodes into a matrix by further replication/partition. The nodes in LBM are mapped into the underlying DHT by hashing the internal labels as well as the matrix coordinates.

Chen et al. [8] suggested a framework for range indexing and proposed various strategies for mapping tree-based index structures into DHTs.

Tanin et al. [20] superimposed the quad tree over the DHT for spatial indexing and querying. Each quad tree node is mapped into the DHT by hashing its centroid. While this paper focuses on one-dimensional data indexing, proposed LIGHT scheme can nevertheless be extended to multidimensional data indexing by employing, for instance, dimension reduction techniques through space-filling curves.

Join queries have attracted considerable research attention in P2P database systems [18]. While focusing on different types of they generally allocate data records by hashing both the names and values of join attributes, and aim to map the joining records to the same DHT node. For these P2P database systems, LIGHT can be seamlessly integrated by indexing the join columns to support general range-based joins, since the essence of such joins consists of range queries in a two-level nested loop.

Gupta et al. [16] applied LSH to DHT-based range indexing and provided approximate range query answers.

For keyword search, Joung et al. [22] proposed a novel indexing scheme, in which uniform hashing is replaced with Bloom filtering, and the underlying overlay is modeled as a multidimensional hypercube.

.

3. METHODOLOGY

3.1 Light Index Structure

Project describes the LIGHT index structure and its mapping strategy to the underlying DHT. Project remarks that LIGHT is proposed to support complex queries over some existing DHTs, while exact-match queries can be directly and efficiently answered by the existing DHT infrastructure.

3.2 Space Partition Tree

As the name implies, the space partition tree (or simply partition tree for short) recursively partitions the data space into two equal-sized subspaces until each subspace contains fewer than Θsplit data keys. Only leaf nodes store data records (or just data entries with pointers pointing to actual data records). Project remark that here a space is always equally partitioned, regardless of the data distribution. This strategy makes the space indexed by each node known globally, which is essential to distributed query processing.

3.3 Local Tree Summarization

Recall that data records are stored in leaf nodes, project need to map only leaf nodes to the underlying DHT. On the other hand, a bare leaf node lacks the knowledge of the overall tree structure, which, as project will see, is critical to complex query processing. Thus, project propose a distributed data structure, termed leaf bucket, to store data records and summarize the partition tree's structural information. Each leaf bucket corresponds to a leaf node in the tree. As illustrated in Fig. 3a, a bucket consists of two fields, leaf label, which maintains the label λ of the leaf node, and record store, which keeps all data records of the leaf node.

// Binary Search Algorithm//

```
Step 1:  μ ← binary-convert(δ)
Step 2:  lower ← 2, upper ← D + 1
Step 3:  while lower ≤ upper do
Step 4:  mid ← (lower+upper)/2
Step 5:  x ←μ.prefix(mid)
Step 6:  bucket_label ←DHT-get(fn(x))
Step 7:  if bucket_label=NULL
         then
         {a failed DHT-get}
Step 8:  upper ← fn(x).length
Step 9:  else if
           bucket_label covers δ then
          {reach the target leaf bucket}
           return fn(x)
Step 10: else
            {x is an ancestor of the target leaf node}
             lower ← fnn(x,μ).length
             return NULL
```

4.2 PROPOSED METHODOLOGY

4.2.1 Complex Queries

In this section, project discuss the processing of various complex queries over the LIGHT index, including range queries, min/max queries, and k-NN queries.

4.2.2 Range Queries

Given two bounds, l and u, a range query returns all data records whose keys fall in the range of [l, u). Thanks to the local tree, LIGHT can support range query processing at near-optimal cost. To illustrate how it works, project start with a simple case.

4.2.2.1 Simple Case

The query issuer happens to be the leaf bucket containing one of the range bounds. Without loss of generality, project assume that it is the lower bound l. As explained earlier, the leaf bucket can construct a local tree, as illustrated in Fig. 6. This figure shows the lower bound leaf λ(l) and all its right neighboring sub trees, denoted by 1, 2, ... In general, the sub tree i covers the data space [pvi, pvi)1), where partition value pvi is the lower bound of the space covered by i. Further denote the right branch nodes by β1, β2,. . . , which can be inferred based merely on the knowledge of λ(l).

4.2.2.2 General Case

The general case, the query issuer can be any leaf bucket. As described in Algorithm 2, after receiving the range query R = [l, u), the leaf locally computes the lowest common ancestor that covers R, abbreviated as LCA. It then forwards the query by a DHT-lookup of fn(LCA). Project discusses three possible cases: 1) The DHT-lookup has failed (line 3), implying that range R is so small that a single leaf completely covers it. The range processing is reduced to a lookup operation. 2) The returned leaf bucket overlaps the query range, implying one range bound must be in this leaf bucket. This is the simple case project discussed above. 3) The returned leaf bucket does not overlap the query range. In this case, the query range is subdivided and, respectively, forwarded to LCA's children, namely LCA0 and LCA1. Note that each of the leaves named to LCA0 and LCA1 must cover one bound of the corresponding sub range. Thus, the processing of both subsequent queries can follow the simple-case strategy.

4.2.2.3 Complexity

The query range is distributed on B leaf buckets. Project here consider only the case where B >= 2 (i.e., Cases 2 and 3 discussed in the last section). In general forwarding, there is at most one DHT-lookup that returns a leaf bucket not overlapping the range. Moreover, as explained in the procedure of each recursive forwarding, there is at most one failed DHT-lookup. Therefore, a total of three extra DHT-lookups can possibly occur, that is, the LIGHT-based range query costs at most B + 3 DHT-lookups, which is close to the optimal performance.

4.2.3 Range Queries with Look ahead

To further reduce the query latency, propose a parallel processing algorithm. The basic idea is that each recursive forwarding in the range query looks one step ahead. That is, for each branch node βi (i = 1, 2, . . . , k) in Fig 6a, the bucket λ forwards the query not only to fn(βi) but also to βi. By this means, each recursive forwarding can explore the neighboring sub tree by two levels (instead of one level as in the original algorithm). Therefore, total latency can be reduced by a factor of two. However, the look ahead can increase the number of DHT-lookup failures, typically from 3 to B/2. This is because in the worst case each look ahead may result in a DHT-lookup failure. As such, the look ahead strategy trades bandwidth overhead for shorter query latency. In general, if project look h steps ahead, the average latency can be reduced by a factor of h + 1, while the number of DHT-lookups is increased by h times. In practice, the user can tune the parameter of h based on his/her performance preferences.

4.2.4 Min/Max Queries

The min (max) query returns the smallest (largest) data key in the data set. Interestingly, LIGHT supports processing a min/max query at constant cost, owing to its nice naming function. More specifically, the query complexity is one DHT-lookup only. In LIGHT, a DHT-lookup of # returns the smallest key, whereas a DHT-lookup of #0 returns the largest key.

4.2.5 K-NN Queries

Given a data key δ and an integer k, the k-NN query returns the k-nearest data keys to δ. LIGHT supports k-NN query processing by a LIGHT lookup of δ, followed by a sequential leaf traversal. Specifically, after the bucket covering δ is located, a bidirectional leaf traversal is set off simultaneously toward the left and the right. Without loss of generality, project focuses on the traversal toward the right. The packet in the leaf traversal carries a parameter unf, which an integer is indicating how many keys still need to be found. It is initiated to k and at any time, unf ≤ k. suppose bucket b receives a k-NN query message of unf and data key δ. Bucket b then returns the results directly to the query issuer (via a physical hop since the query issuer's address can be known from the packet header). The query issuer will update the value of unf according to the current result set and notify bucket b of the new unf0. If the new unf' is still bigger than 0, meaning that the current result set is not yet filled up, bucket b further forwards the query to its immediate right neighbor. This is quite similar to the forwarding to βi in the range query. A k-NN query traversing B buckets incurs at most 1:5B DHT-lookups since in the worst case 50 percent of DHT-lookups might fail (e.g., the hop from #0011 to #0100 always succeeds but the one from #0010 to #0011 can fail).

4.2.6. B+ Tree Based Improved Light

The B+ tree based improved LIGHT perform a search for a Query Q (record r) follows pointers to the correct child of each node until a leaf is reached. Then, the leaf is scanned until the correct Query is found.

```
Function search (Query Q)
        u := root
  While (u is not a leaf) do
        Choose the correct pointer in the node
Move to the first node following the pointer
        u := current node
        Scan u for Q
```

// Peer Node Insertion//

Step 1: *Perform a search to determine what bucket the new query.*
Step 2: *If the bucket is not full, add the query.*
Step 3: *Otherwise, split the bucket.*
Step 4: *Allocate new leaf and move half the bucket's elements to the new bucket.*
Step 5: *Insert the new leaf's smallest key and address into the parent.*
Step 6: *If the parent is full, split it too.*
Add the middle key to the parent node.
Step 7: *Repeat until a parent is found that need not split.*
Step 8: *If the root splits, create a new root which has one key and two pointers.*

// Peer Node Deletion//

Step 1: Start at root, find leaf L where entry belongs.
Step 2: Remove the entry.
Step 3: If L is at least half-full, done!
If L has entries less than it should,
Try to re-distribute, borrowing from sibling
(adjacent node with same parent as L).
Step4: If re-distribution fails, merge L and sibling.
Step 5: If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
Step 6: Merge could propagate to root, decreasing height.

5. EXPERIMENTATION & RESULTS

The leaves (the bottom-most index blocks) of the B+ tree are linked to one another in a linked list. It makes range queries or an (ordered) iteration through the blocks simpler and more efficient (upper bound achieved even without this addition). This does not substantially increase space consumption or maintenance on the tree. The major significant advantages of a B+-tree over a LIGHT is that, since not all pseudo columns present in the leaves, such an ordered linked list cannot be constructed. B+-tree is thus particularly useful as a query index, where the query typically resides on peer nodes, as it allows the B+-tree to actually provide an efficient structure for maintaining the query itself

If a query storage system has a block size of B bytes, and the keys to be stored have a size of k, most efficient B+ tree is one where $b = (B / k) - 1$. In practice there is a little extra space taken up by the index blocks. Having an index block which is slightly larger than the storage system's actual block represents a significant performance decrease, therefore erring on the side of caution is preferable. If nodes of the B+ tree are organized as arrays of elements, then it take a considerable time to insert or delete an element as half of the array will need to be shifted on average. To overcome this problem, elements inside a node can be organized in a binary tree or a B+ tree instead of an array. B+ trees is used for queries stored in peer nodes about their range queries in the neighboring peers. In this case a reasonable choice for block size would be the size of peer node's cache size. The block size if few times larger than the peer's cache size, it delivers better performance if cache pre-fetching is used.

Project implemented LIGHT in Java. The total number of code lines is 2,200 (including LIGHT, DST, and PHT), which demonstrates the simplicity of developing an over- DHT indexing scheme. In the experiments, LIGHT, DST, and PHT were run over the Bamboo DHT [17], a ring-like DHT that has good robustness and is now widely deployed in the Open DHT project. Our whole system was deployed in a LAN environment consisting of more than 20 computers (or peers). Both real data and synthetic data were tested. For the real data, project used the DBLP data set, which contains the publications listed in the DBLP Computer Science Bibliography.

The other names were converted to a floating number in the domain of [0, 1] and used as the data keys. By filtering out duplicate author names, project obtained a DBLP data set containing approximately 250,000 distinct data keys. Project further divided the whole data set into five smaller data sets with 50,000 data keys each. The experiments were conducted against all the five small data sets; the average performance is reported here. To evaluate the scalability of the indexing schemes, project also used two synthetic data sets, uniform and gaussian, with sizes varying from 500,000 to 8,000,000.

The data keys in the uniform data set were randomly generated in [0, 1], while the data keys in the gaussian data set follow a gaussian distribution with a mean of 1/2 and a standard deviation of 1/ 6 , which guarantees that about 97 percent of the keys will fall in [0, 1]. For performance testing on the synthetic data, project repeated each experiment over 30 times and reports the average results.

5.1 Structural Properties

Project examines the structural properties of the LIGHT index, including average leaf depth, number of leaf nodes, and bucket utilization. Bucket utilization is defined to be the ratio of the number of records stored in a leaf bucket to the bucket capacity ϴ split. Project measures these properties after project inserted 50,000 data keys into the LIGHT index. The performance trends when ϴ split is varied from 50 to 1,000. When ϴ split. grows large, both the average leaf depth and the number of leaf nodes decrease since a large ϴ results in leaves containing more keys and thus fewer leaf nodes. Comparing the three data sets under testing, DBLP has more and deeper leaf nodes.

This is because the data distribution in DBLP is highly skewed, which makes the index tree very unbalanced. Most leaf nodes for the uniform data set have a depth of 13 or 14, whereas the depth of the leaf nodes for DBLP varies from 10 to 25.The bucket utilization as a function of ϴ split. As expected, the bucket utilization for the DBLP data set is lowest due to the skewness of data distribution. The bucket utilization for the synthetic data sets, especially the uniform data set, fluctuates as ϴ split increases, owing to the characteristic of the space partition tree.

5.2 Lookup Performance

The efficiency of looking up a key in the index, project compares LIGHT with PHT with varying data set sizes. Note that the lookup operations in both LIGHT and PHT have a parameter D, the maximum leaf depth. To make a fair comparison, D is always set to the actual maximum tree depth for the data set under testing. The splitting threshold ϴ split is fixed at the default value 100. For each experiment, project conduct 1,000 lookups for the keys uniformly distributed in [0, 1] and record the average number of DHT-lookups per lookup operation.

In general, as expected, the number of DHT-lookups increases as the data set grows. For the DBLP and gaussian data sets, LIGHT outperforms PHT by 35 percent on average. For the uniform data set, the performance curve of PHT exhibits a zigzag shape. This is because most leaf buckets reside in the deepest two levels of the tree. As the data set size is increased, the numbers of leaf buckets on these two levels are increased in turn, for which the binary search gets a fluctuating lookup performance.

5.3 Index Maintenance Performance

Project now evaluates the index maintenance performance under data insertions and deletions. In the following, first compare LIGHT with PHT for the leaf split cost. Then compares the overall index maintenance performance among LIGHT, PHT and DST.

5.3.1 Leaf Split Costs

It first measure the value of α for LIGHT, that is, the ratio of data records moved to remote peers during a leaf split. To evaluate it, project continuously inserts data into the LIGHT index and record the average value of α for the leaf splits. The average α remains almost constant under different data set sizes for the uniform and gaussian data sets. For the DBLP data set, the average α fluctuates a little bit when the data set size is smaller than 15,000 but becomes stable as the size of the data set increases. This is mainly because of the irregular distribution of DBLP data. The average α fairly approaches the value of 0.5, which is consistent with our previous analysis .Next project compare LIGHT with PHT for the leaf split performance. Project continuously inserts data into LIGHT and PHT and records the cumulative split costs. Recall that our leaf split involves data-movement costs and DHT-lookup costs. Project measures them separately in each experiment. Total data-movement costs slightly decrease as Ɵsplit increases, while the number of DHT-lookups is inversely proportional to Ɵ. The reason is that a larger Ɵsplit results in fewer split operations. Comparing LIGHT with PHT, LIGHT improves PHT by 50 percent for data-movement costs and 75 percent for DHT-lookup costs, which conforms to our previous analysis. To further test the scalability, project conduct experiments on the synthetic data sets with varying data set sizes. A similar performance improvement can be observed under different data set sizes for both the uniform and gaussian data sets.

5.3.2 Performance under Data Insertions

It evaluates performance under data insertions, which includes the costs incurred by both data insertion and leaf split. The same experimental settings are chosen as with the leaf split experiments. Project can see that DST incurs a cost higher than LIGHT and PHT by an order of magnitude. This is because DST employs data replication. More specifically, each insertion in DST needs to look up all the ancestors of the leaf and insert the data into the unsaturated ancestors, which typically amplifies the insertion cost by a factor of D.

Comparing LIGHT and PHT, LIGHT still outperforms PHT by about 40 percent for data-movement costs and 30 percent for DHT-lookup costs. This is because LIGHT achieves more efficient lookup and leaf split operations during the insertion process. It is also interesting to observe that the relative performance of LIGHT, PHT, and DST is quite insensitive to the data distribution.

5.3.3 Performance under Data Deletions

The next study performance under data deletions. The experiments proceed in three phases, the growing phase, in which only data insertion is allowed, the steady phase, in which data insertions and deletions are randomly performed, and the shrinking phase, in which data is deleted from the P2P index until it is contracted into a single root. Recall that the leaf merge operation requires a threshold Ɵ merge In the experiments, Ɵmerge is set to 0.5 . Ɵ split and 0.2 . Ɵ split. The data-movement costs remain relatively stables in the steady phase, implying that the split or merge operations rarely happen during this phase. This is because random insertions and deletions may cancel out each other's effects. Throughout the whole process, the cost of LIGHT remains half that of PHT, and they are both insensitive to the value of merge. Similar performance results are also observed for the uniform and gaussian data sets.

5.3.4 Range Query Performance

It evaluates the query processing performance for range queries. The evaluation is in terms of two aspects, time latency and bandwidth costs. The former is captured by the paralleled steps of DHT-lookups, while the latter is captured by the total number of DHT-lookups. Recall that project proposed two LIGHT range query algorithms, the basic one and the one with lookahead. PHT also has two range query algorithms, denoted as PHT (sequential) and PHT (parallel), respectively. Project compares these four range query algorithms together with DST. 10 Among the five algorithms, LIGHT (basic) achieves the lowest bandwidth (though not quite visible in the figure), while PHT (sequential) requires a bandwidth slightly higher than LIGHT (basic). As discussed earlier, their performance nearly approaches the optimum, that is, the number of DHTlookups equals the number of target leaf buckets.

The bandwidth costs of PHT (parallel) and DST are twice that of LIGHT (basic) because they both incur internal node traversal when processing range queries. The bandwidth costs of LIGHT (lookahead) are approximately 50 percent higher than the optimal bandwidth, which again conforms to our previous complexity analysis. Without leveraging parallelism, PHT (sequential) incurs extremely high latency. Although parallelism is employed in PHT (parallel) and DST, they still suffer from data skewness for which the deepest leaf node dominates the whole query process. For the scalability test on the synthetic uniform and gaussian data sets, a similar result is found. The only exception here is that LIGHT (basic) incurs a slightly higher latency than DST because the skewness is much lower in the synthetic data and DST suits such unskewed distribution. In summary, LIGHT (basic) outperforms all others in terms of bandwidth costs and achieves quite good time latency, just behind LIGHT (look ahead). LIGHT (look ahead) trades bandwidth for time latency, which makes its time latency the shortest. PHT(sequential) achieves quite efficient bandwidth costs but incurs extremely high latency. PHT(parallel) and DST both incur the highest bandwidth costs, but their latency is not yet the most efficient.

6. CONCLUSION

The proposed work presented an improved version of LIGHT scheme by adapting B+ tree indexing structure to the existing LIGHT scheme for efficient query exchange between peer nodes in dynamic internet. The B+ tree indexing added to the LIGHT sorted queries in the peer network with more efficient insertion, retrieval and removal of leaf nodes in the hash structure by a pseudo column. Proposed version of improved LIGHT is dynamic, multilevel index, with maximum and minimum bounds on the number of pseudo column values in each index segment. The maximum and minimum bounds derived in the proposed scheme reduces the pruning time along with memory usage for index storing compared to that of the existing LIGHT scheme. In the B+ tree, in contrast to existing LIGHT tree, all records are stored at the leaf level of the tree, only pseudo column values are stored in interior nodes. The prime concern of improved LIGHT version is in storing queries for efficient retrieval in a block-oriented storage context. Unlike binary search trees, B+ trees have very high fan out which reduces the number of I/O operations required to find an element in the query search tree of the peer to peer networks. Experimental results shows that the performance of improved LIGHT with the B+ tree indexing structure shows better throughput and minimal I/O operation compared to that of existing LIGHT and DHT scheme. The overheads of the query exchange on dynamic peer networks are also reduced to nearly 40% in improved LIGHT scheme compared to existing LIGHT.

REFERENCES

[1] Andrzejak A. and Xu Z, "Policies for Efficient Data Replication in *P2P* Systems" Proc. IEEE Int'l Conf. Peer-to-Peer Computing (P2P), Pages : 33-40, 2015.

[2] Aspnes J. and Shah G, "Skip Graphs," Proc. Symp. Discrete Algorithms (SODA), Pages : 235-249, 2015.

[3] Bawa M., Condie T., and Ganesan P, "Lsh Forest: Self-Tuning Indexes for Similarity Search," Proc. World Wide Web (WWW), Pages : 163-175, 2015.

[4] Bharambe A.R, Agrawal M. and Seshan S, "Mercury: Supporting Scalable Multi-Attribute Range Queries," Proc. 2014 Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. and ACM SIGCOMM Computer Comm. Rev., Pages : 31-42, 2014.

[5] Bridges W.G. and Toueg S, "On the Impossibility of Directed Moore Graphs," J. Combinatorial Theory, Series B, vol. 29, no. 3, Pages : 339-341, 2014.

[6] Cai M. and Frank M.R, "RDFpeers: A Scalable Distributed RDF Repository Based on a Structured Peer-to-Peer Network," Proc. World Wide Web (WWW), Pages : 460-462, 2014.

[7] Chawathe Y., Ramabhadran S., Ratnasamy S., LaMarca A., Shenker S., and Hellerstein J.M, "A Case Study in Building Layered DHT Applications," Proc. 2005 Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM),Pages : 374-399, 2015.

[8] Chen L., Candan K.S., Tatemura J., Agrawal D., and Cavendish D, "On Overlay Schemes to Support Point-in-Range Queries for Scalable Grid Resource Discovery," Proc. IEEE Int'l Conf. Peer-to- Peer Computing (P2P), Pages : 23-30, 2005.

[9] Crainiceanu A., Linga P., Gehrke J., and Shanmugasundaram J, "Querying Peer-to-Peer Networks Using P-Trees," Proc. Workshop Web and Databases (WebDB), Pages : 311-324, 2015.

[10] Crainiceanu A., Linga P., Machanavajjhala A., Gehrke J. and Shanmugasundaram J, "P-Ring: An Efficient and Robust P2P Range Index Structure," Proc. ACM SIGMOD, Pages : 359-401, 2015.

[11] Du Mouza C., Litwin W. and Rigaux P, "SD-Rtree: A Scalable Distributed Rtree," Proc. Int'l Conf. Data Eng. (ICDE), Pages : 296-305, 2007.

[12] Fraigniaud P. and Gauron P, "Brief Announcement: An Overview of the Content-Addressable Network D2B," Proc. Principles of Distributed Computing (PODC), Page : 164-167, 2014.

[13] Galanis L., Wang Y., Jeffery S.R., and DeWitt D.J, "Locating Data Sources in Large Distributed Systems," Proc. Very Large Data Bases (VLDB), Pages : 874-885, 2003.

[14] Gao J. and Steenkiste P, "An Adaptive Protocol for Efficient Support of Range Queries in DHT-Based Systems," Proc. IEEE Int'l Conf. Network Protocols (ICNP), Pages : 143-159, 2015.

[15] Gao J. and Steenkiste P, "Efficient Support for Similarity Searches in DHT-Based Peer-to-Peer Systems," Proc. Int'l Conf. Comm. (ICC), Pages : 1867-1874, 2007.

[16] Gupta A., Agrawal D., and Abbadi A.E, "Approximate Range Selection Queries in Peer-to-Peer Systems," Proc. Conf. Innovative Data Systems Research (CIDR), 2003.

[17] Han D., Shen T., Meng S. and Yu Y, "Cuckoo Ring: Balancing Workload for Locality Sensitive Hash," Proc. IEEE Int'l Conf. Peerto- Peer Computing (P2P), Pages : 49-56, 2006.

[18] Huebsch R., Hellerstein J.M., Lanham N., Loo B.T., Shenker S., and Stoica I, "Querying the Internet with Pier," Proc. Very Large Data Bases (VLDB), Pages : 321-332, 2003.

[19] Jagadish H.V., Ooi B.C., and Vu Q.H, "Baton: A Balanced Tree Structure for Peer-to-Peer Networks," Proc. Very Large Data Bases (VLDB), Pages : 661-672, 2005.

[20] Jagadish H.V., Ooi B.C., Vu Q.H., Zhang R. and Zhou A, "VBITree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes," Proc. Int'l Conf. Data Eng. (ICDE), Page : 34, 2006.

[21] Joung Y.J. and Yang L.W, "KISS: A Simple Prefix Search Scheme in P2P Networks," Proc. Workshop Web and Databases (WebDB), 2006.]

[22] Joung Y.J., Fang C.T., and Yang L.W, "Keyword Search in DHTBased Peer-to-Peer Networks," Proc. Int'l Conf. Distributed Computing Systems (ICDCS), Pages : 339-348, 2005.

[23] Li D. Lu X., and Wu J, "Fissione: A Scalable Constant Degree and Low Congestion DHT Scheme Based on Kautz Graphs," Proc. IEEE Int'l Conf. Computer Comm. (INFOCOM), Pages : 1677-1688, 2005.

[24] Liang J. and. Nahrstedt K, "Randpeer: Membership Management for QoS Sensitive Peer-to-Peer Applications," Proc. IEEE Int'l Conf. Computer Comm. (INFOCOM), 2006.

[25] Loguinov D., Kumar A., Rai V., and Ganesh S, "Graph-Theoretic Analysis of Structured Peer-to-Peer Systems: Routing Distances and Fault Resilience," Proc. 2003 Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM), Pages : 395-406, 2014.

[26] Malkhi D., Naor M., and Ratajczak D, "Viceroy: A Scalable and Dynamic Emulation of the Butterfly," Proc. Principles of Distributed Computing (PODC), Pages : 183-192, 2012.

[27] Ramabhadran S., Ratnasamy S., Hellerstein J.M., and Shenker S, "Brief Announcement: Prefix Hash Tree," Proc. Principles of Distributed Computing (PODC), Page : 368, 2014.

[28] Ratnasamy S., Francis P., Handley M., Karp R.M., and Shenker S, "A Scalable Content-Addressable Network," Proc. 2001 Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM), Pages : 161-172, 2014.

[29] Reynolds P. and Vahdat A, "Efficient Peer-to-Peer Keyword Searching," Proc. Middleware, Pages : 21-40, 2003.

[30] Rhea S.C., Geels D., Roscoe T., and Kubiatowicz J, "Handling Churn in a DHT," Proc. USENIX Ann. Technical Conf. (ATC), Pages : 127-140, 2004.

[31] Rowstron A.I.T. and Druschel P, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," Proc. Middleware, Pages : 329-350, 2013.

[32] Sahin O.D., Gulbeden A., Emekci F., Agrawal D., and Abbadi A.E, "PRISM: Indexing Multi-Dimensional Data in P2P Networks Using Reference Vectors," Proc. 13th Ann. ACM Int'l Conf. Multimedia (MM), Pages : 946-955, 2012.

[33] Stoica I., Morris R., Karger D.R.,. Kaashoek M.F, and Balakrishnan H, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," Proc. 2003 Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM), Pages : 149-160, 2015.

[34] Tanin E., Harwood A., and Samet H, "Using a Distributed Quadtree Index in Peer-to-Peer Networks," Very Large Data Bases J., vol. 16, no. 2, Pages : . 165-178, 2014.

[35] Zhao B.Y., Kubiatowicz J. and Joseph A.D, "Tapestry: A Fault- Tolerant Wide Area Application Infrastructure," Computer Comm. Rev., vol. 32, no. 1, Page : 81, 2012.

[36] Zheng C., Shen G., Li S., and Shenker S, "Distributed Segment Tree: Support of Range Query Cover Query over DHT," Proc. Fifth Int'l Workshop Peer-to-Peer Systems (IPTPS), Feb. 2015.