Two-Level Indexing for High-Dimensional Range Queries in Peer-to-Peer Networks

Lelin Zhang #1, Zhiyong Wang #2, Dagan Feng #*3

School of Information Technologies, University of Sydney, Sydney, Australia

1 lzha1533@usyd.edu.au
2 zhiyong@it.usyd.edu.au
3 feng@it.usyd.edu.au

* *Department of Electronic and Information Engineering, Hong Kong Polytechnic University, Hong Kong, China*

***Abstract*—Supporting complex and efficient lookup queries in peer-to-peer networks is challenging, though simple keyword based lookup queries are well supported by most deployed systems. This paper presents a two-level indexing structure built on Distributed Hash Table (DHT) aiming to support range queries on high-dimensional feature space in peer-to-peer network. Unlike most existing systems, where every node is responsible for a data partition, our design only utilizes a small part of the nodes to manage partitions. These partition nodes form the first level index. The second level index consists of one or more server nodes, which maintains links to each partition node. Additionally, a merge and split mechanism is designed to dynamically adjust the workload among nodes. Experimental results indicate that our system offers promising performance in terms of workload balance in churn networks. The flexibility to work with any DHT and the capability to support multiple feature spaces further make our proposed approach a feasible extension for file sharing networks.**

I. Introduction

Peer-to-peer networks consist of peer nodes simultaneously acting as client and server. Its scalable and robust nature is becoming more and more popular in various applications like file sharing. However, due to its distributed nature, supporting complex and efficient lookup queries in peer-to-peer networks is challenging. While most deployed systems support only simple keyword based lookup queries, some approaches have recently been proposed to extend the query functionality. For instance, Skip Graphs [1] and BATON [2] can support efficient range queries over a single attribute, or one-dimensional feature spaces. Other approaches [3][4][5][6][7] take a step further to multi-dimensional range queries. One straightforward way to achieve this is to project or map the multi-dimensional data into one dimension. SCARP [7], ZNet [3] and SkipIndex [4] all use the concept of Space Filling Curve (SFC) to map multi-dimensional data into one dimension value. Other approaches using similar ideas include Locality Preserving Function in CISS [8] and *tree linearization* in Brushwood [9]. Although this concept yields good results in low-dimensional space, its performance becomes problematic when the dimension is high. As mentioned in [7], they all suffer from the fact that nearby points in native space become increasingly far apart in the mapped 1-D space as feature dimensionality increases.

Supporting high-dimensional range queries is essential for content based Multimedia Information Retrieval. One of the approaches which provide efficient high-dimensional range queries is Multidimensional Rectangulation with Kd-Trees (MURK) [7]. Inspired by the fact that the content-addressable network (CAN) [10] forms a virtual d-dimensional Cartesian coordinate space, MURK maps the feature vector, rather than hash values onto the space. A routing scheme similar to CAN is used to search a range in the feature space. Additionally, MURK tries to use a Kd-Tree approach to partition the space in order to distribute data evenly around nodes. However, in churn networks, where nodes keep joining and leaving and data keep adding and removing, it is difficult to adjust the partitions of a Kd-Tree, which causes a serious workload balancing problem for MURK. Furthermore, since MURK's design is dependent on the underlayer structure, it loses the flexibility to support multiple feature spaces.

In this paper, we propose a novel design aiming to support efficient multi-dimensional range queries in peer-to-peer file sharing networks. The design is a two-level index built above DHT. In contrast to most of the existing approaches that attempt to separate indexing data to every node in the network, our design uses only part of the nodes, called partition nodes. Each partition node manages a partition of the entire feature space, which forms the first-level index. In addition, one or more designated server nodes will store links to all the partition nodes, which forms the second-level index. Answering a range query invokes two stages: firstly the query is sent to a server node to obtain the relative partitions; then the query is sent to corresponding partition nodes to obtain the desired data.

We utilize the Kd-Tree [11] partitioning method, which splits partitions with equal amount of data instead of equal space. In order to keep appropriate workload on each partition node, a split algorithm, which separates the data on one node into two equal halves, is invoked when a node becomes overloaded; a merge algorithm, which combines n adjacent

MMSP'09, October 5-7, 2009, Rio de Janeiro, Brazil.
978-1-4244-4464-9/09/$25.00

partitions into $n-1$ partitions, is invoked when a node becomes idle.

With the above settings, our design achieves the following properties:

- Promising workload balancing and adaptability to changes. With the split and merge mechanism, the workload in every node will be kept in an adequate level. No node will be overloaded even in skewed data or with rapid change in data distribution.
- Flexibility to work with any DHT and multiple feature spaces. Unlike some systems, our design is independent to underlayer structure. This gives us the flexibility to build our system on any DHT, as long as it provides STORE and LOOKUP primitives. Also, we can easily support multiple feature spaces in one network, and add or remove feature space in run-time. Such a property is desirable in file sharing, by which different types of files are being shared and new file types keep emerging.
- Competitive efficiency. Since the system uses a two-level index, any given query can be answered with two DHT lookups. Provided that most existing DHTs can do the lookup in $O(logn)$ time, our system can also answer a multi-dimensional range query in $O(logn)$ time.

The rest sections of this paper are organized as follows: In Section 2 we explain different components of our design in detail. In Section 3 we present some of the evaluation results. Finally the conclusion is given in Section 4.

II. Two-level Indexing

In our design, the feature space is split into orthogonal partitions. Each partition is managed by a randomly selected node (partition node) of the network. Only a small portion of the nodes of the network will be selected as partition node because the computation power is redundant. An overloaded partition node can easily find a regular node and hand over half of its data by SPLIT mechanism.

Since the partitions are dynamically changing, a server node is needed to store links to the partition nodes. The server node must be determined by a globally agreed method, such that every node in the network can find the server node. One method is applying the consistent hash function with the name of the feature space. For example, if the underlying DHT is a Kad network, which uses SHA-1 as hash function, then the server node's ID for mp3 file may be SHA-1("mp3"), while jpg file may be SHA-1("jpg").

As every query is first sent to a server node, the server node may become the bottleneck of the network. Therefore multiple server nodes are defined to share the workload. In this case, every server node keeps exactly the same partition index. A query can be answered by any server node, whereas the split and merge should notify every server node. As shown in Section 3, this strategy can efficiently share the workload in our context.

Note that the mappings to both partition node and server node are virtual, i.e. several partitions and / or the server index can be managed at one node, as long as it is not overloaded. This facilitates the support of multiple feature spaces in one network.

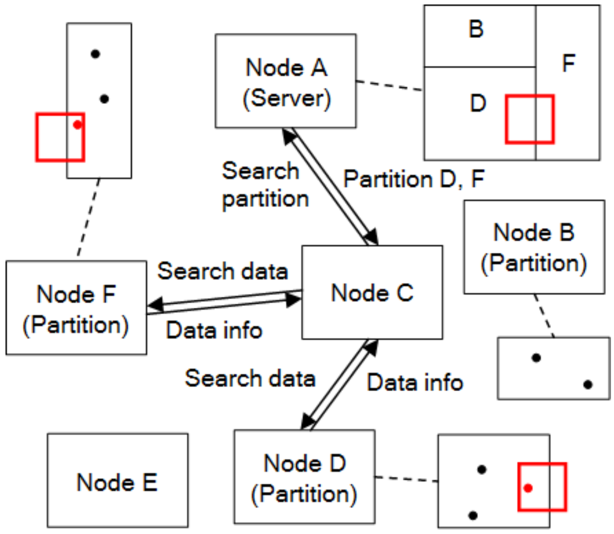

Fig. 1. Illustration of a range query in the two-level indexing scheme.

The whole structure is built on DHT. It utilizes the DHT STORE and LOOKUP interface to transfer message and data. As illustrated in Figure 1, answering a query consists of two stages. Firstly the query is sent to the server node A, getting the relative partition nodes. Then the request node sends the query to relative partition nodes D, F to get the matching result.

Node joining and leaving are automatically handled by the underlayer DHT. Inserting a data item to a partition node may overload the corresponding node. In this case, the node will perform a SPLIT operation and hand over half of its data to a randomly selected node. Similarly, if a partition node becomes idle because of data removing, it will perform a MERGE operation and hand over all its data to its neighboring partitions.

The SPLIT and MERGE operations keep an appropriate workload on partition nodes. The algorithms are optimized to minimize node communications and node states, which is desirable in peer-to-peer environment.

A. Split Operation

The SPLIT operation is performed by an overloaded partition node, which splits the partition into two partitions with equal amount of data. As a result, a randomly selected node will be become a new partition node and take over half of the data. The split is identical to a Kd-Tree split except that a randomly selected split dimension is used, since we don't maintain a tree structure.

To make sure a merge operation can be performed later, an additional mark called EdgeGeneration on each splitting edge is used. On each split, the current "youngest" edge of the old partition is found, and the new splitting edge is marked as one generation younger than it.

The SPLIT operation is described in Algorithm 1. As an example, Figure 2 shows the splitting of partition E. As partition E has three edges with EdgeGeneration 0, 1 and 2, the youngest edge is edge between D and E, which EdgeGeneration is 2, thus the new edge will have an EdgeGeneration 3.

Algorithm: SPLIT(*P*)
Input: A *d*-dimensional partition *P*
Output: Two partitions *P*[1], *P*[2], as the result of split
1. Let *g* be the largest (youngest) EdgeGeneration of *P*.
2. Perform a Kd-Tree split on randomly selected dimension.
3. Mark the EdgeGeneration of the splitting edge of *P*[1] and *P*[2] as *g* + 1.
4. Update the corresponding neighbor information.
5. Notify all the neighbors about the split.

Algorithm 1 SPLIT

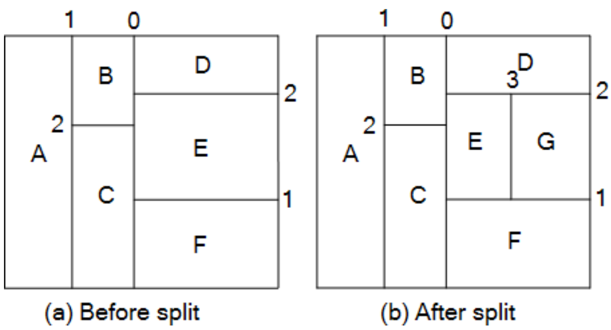

Fig. 2. Illustration of the SPLIT algorithm.

B. Merge Operation

The MERGE operation described in Algorithm 2 is performed by an idle partition node, and it merges the partition with some of its neighbors thus reduce the total number of partitions by 1. As a result, the data of the current partition is handed over to its neighbors. The EdgeGeneration information stored before is used to direct the merge. Basically, the partition which initiates a merge operation (merger) will find neighbors of its youngest edge (mergees), and split its partition to all the mergees.

Figure 3 shows the merge process of a partition D. D's youngest edge is the edge between D and E, G, hence the mergee of D is partitions E and G. Therefore, D split itself into two partitions which merge with E and G correspondingly. The result after merge is shown at Figure 3(b). In this case, 3 partitions are merged into 2.

III. EVALUATION

To evaluate our design, we simulate the network evolution with time frame based settings: in each time frame there are certain probabilities that a node is joining or leaving, and

---

Algorithm: MERGE(*P*)
Input: A *d*-dimensional partition *P*
Output: *m* partitions {*P*[1], *P*[2], …, P[*m*]}, as the result of merge
1. Find the edge of *P* which has largest EdgeGeneration, let the edge be *E*.
2. Find all the neighboring partitions on edge *E*, let them be {*P*[1], *P*[2], …, *P*[*m*]}.
3. Split *P* according to the neighbors, let the corresponding result be {*P'*[1], *P'*[2], …, *P'*[*m*]}.
4. Merge the corresponding partitions, i.e. merge *P*[1] with *P'*[1], *P*[2] with *P'*[2], …, *P*[*m*] with *P'*[*m*].
5. Update the neighbor information.
6. Notify all the neighbors about the merge.

Algorithm 2 MERGE

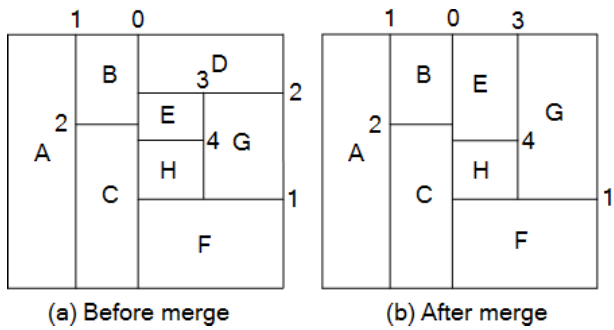

Fig. 3. Illustration of the SPLIT algorithm.

that a data item is adding or removing. In our experiment the probabilities of node joining / leaving are 0.1 while data item adding / removing are 0.8 (we notice that different probabilities settings may "stretch" or "shrink" the change over time, but will not affect the overall trend).

The average workload of a node is set to 100. This number is based on workload data of 5 popular ed2k servers[1] , which shows an average of 82.36 files per node. The process capacity of a node is 1000 (thus a SPLIT operation will be invoked if a partition node holds more than 1000 data items). Also, the minimum workload of a node is set to 200 (thus a MERGE operation will be invoked if a partition node holds fewer than 200 data items).

The number of dimensions of feature space is set to 10. The data is randomly generated with normal distribution.

A. Dynamic Load Balance

A 2-stage experiment is conducted in order to compare the load balance performance with MURK:

- Initialization (INIT): We set up a network with 100 nodes and 10,000 data items. The data obey a normal distribution with mean = 0. For MURK, a Kd-Tree construction strategy is used to ensure that the workload is balanced at the beginning.
- Stage 1 – same distribution (SDIST): The network runs for 10,000 time frames to simulate the dynamic changes,

[1] http://www.emule-project.net

i.e. nodes joining / leaving and data inserting / removing. The new data items added follow the same distribution as the those in initialization.

- Stage 2 – different distribution (DDIST): At this stage, the network runs for 100,000 time frames. New data items added obey a normal distribution with mean = 100, which simulates a dramatic change of data distribution.

At the end of each stage, the number of partitions and the number of data items inside each partition are examined. The experimental results are summarized in Table 1, where N is the number of partitions.

TABLE I
PERFORMANCE OF WORKLOAD BALANCING

	N	Mean	Standard Deviation	Min.	Max.	Range
INIT[Proposed]	16	625.000	24.109	582	665	83
INIT[MURK]	100	100.000	34.924	78	156	78
SDIST[Proposed]	16	627.125	25.809	590	681	91
SDIST[MURK]	117	85.667	153.732	0	911	911
DDIST[Proposed]	16	636.813	117.479	248	778	530
DDIST[MURK]	206	48.709	180.060	0	1261	1261

As MURK maps every node to a partition, it has far more number of partitions than our method, thus the average amount of data in each partition is much smaller than our method. However, as indicated by the standard deviation, while the network evolves, workloads on MURK become seriously unbalanced. As shown in Figure 4, at the end of the experiment, most of the partitions in our system hold around 500 to 800 data items. In contrast, most of the partitions in MURK holds 0 to 100 data items (in fact, most of these partitions holds no data), which results in increased query cost and decreased data locality. Even worse, 3 partitions in MURK contain more than 1,000 data items, which means overloaded with our experiment setting.

B. Change Adaptation

Figure 5 shows the change of workload variance during different experiment stage.

In stage 1, the data distribution remains unchanged. In our system, nodes joining and leaving do not affect the partitions, thus good load balance is maintained as expected. In MURK, every node joining or leaving forces a random partition to split or merge. Therefore, its workload soon becomes unbalanced.

In stage 2, the data distribution changes dramatically. In our system, as data of new distribution arrives, the old partitioning can not separate the space with equal load any longer. Soon several partitions near the hotspot become overloaded, and split into smaller partitions. As the old data keeps removing, the partitions among old hotspot will gradually become empty, and merged into larger partition. That's why the workload variance increases at the beginning and gradually decreases. Note that after 40,000 time frames, our load balance performance becomes stable, which shows that our partitioning adapts to new data distribution, and no more merge and split operations are performed. Figure 6 illustrates how the partitioning changes adapt to the change of data distribution with hotspots circled with dashed line. In MURK, the performance is unpredictable because it provides no load balancing mechanism.

C. Server Node Workload

Let d be the average data workload of a node, c be computation capacity of a node, and we define the workload ratio $r = d/c$. With a network of n nodes, at least $r \times n$ partition nodes is needed.

Assume the number of queries is proportional to the amount of data, the query workload of server nodes will scale as $O(r \times n)$. With multiple server nodes, queries can be answered by any server node, thus the query workload of a single server node will be $O(r \times n/s)$, where s is the number of server nodes.

However, the split and merge notifications, which need to be sent to every server nodes, are not shareable. Fortunately, this overhead cost is trivial compared to queries. In fact, even in the most dynamic environment in our experiment (0 to 40,000 time frames in stage 2), the network performs only 16 Split and 16 Merge operations.

In summary, the workload on server nodes scales roughly as $O(r \times n/s)$, which is scalable in a file sharing network, where the workload ratio r is small.

IV. CONCLUSION

We presented a new distributed data structure built on DHTs so that range queries on high-dimensional feature spaces can be supported. A merge and split mechanism is designed to dynamically adjust the load between nodes. Experimental results indicate that our system offers good load balance performance in dynamic environment. Our design has provided a feasible solution for adding Multimedia Information Retrieval functions in practical complicated peer-to-peer file sharing applications.

ACKNOWLEDGEMENT

The work presented in this paper is partially supported by ARC and Hong Kong Polytechnic University grants.

REFERENCES

[1] J. Aspnes and G. Shah, "Skip graphs," *ACM Transactions on Algorithms*, vol. 3, no. 4, p. 37, 2007.

[2] H. V. Jagadish, B. C. Ooi, and Q. H. Vu, "Baton: a balanced tree structure for peer-to-peer networks," in *VLDB '05: Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 661–672.

[3] Y. Shu, B. C. Ooi, K.-L. Tan, and A. Zhou, "Supporting multi-dimensional range queries in peer-to-peer systems," in *Peer-to-Peer Computing, IEEE International Conference on*. IEEE Computer Society, 2005, pp. 173–180.

[4] C. Zhang, A. Krishnamurthy, and R. Y. Wang, "Skipindex: Towards a scalable peer-to-peer index service for high dimensional data," Department of Computer Science, Princeton University, Tech. Rep., May 2004.

[5] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: supporting scalable multi-attribute range queries," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 353–366, 2004.

[6] M. Batko, D. Novak, F. Falchi, and P. Zezula, "On scalability of the similarity search in the world of peers," in *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*. New York, NY, USA: ACM, 2006, p. 20.

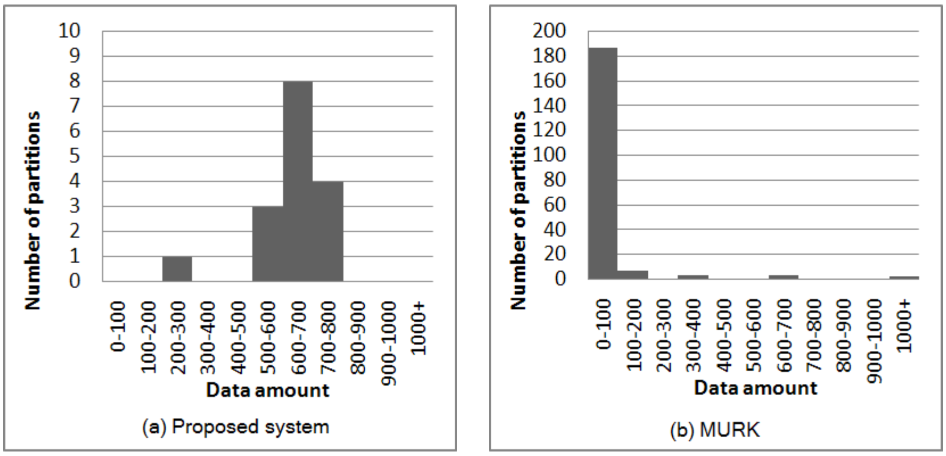

Fig. 4. Illustration of workload distribution after stage 2.

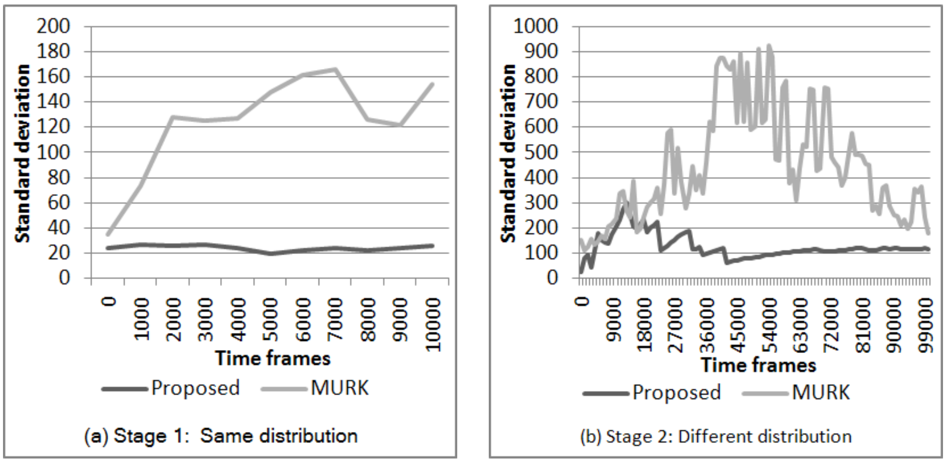

Fig. 5. Illustration of the change of workload variance in different stages.

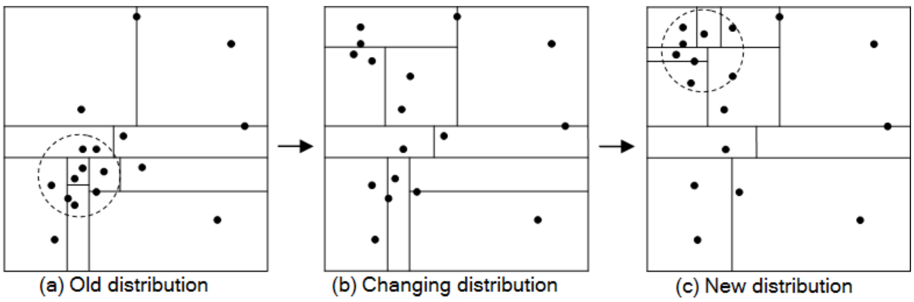

Fig. 6. Evolution of partitioins in response to the change of data distribution.

[7] P. Ganesan, B. Yang, and H. Garcia-Molina, "One torus to rule them all: multi-dimensional queries in p2p systems," in *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*. New York, NY, USA: ACM, 2004, pp. 19–24.

[8] J. Lee, H. Lee, S. Kang, S. M. Kim, and J. Song, "Ciss: An efficient object clustering framework for dht-based peer-to-peer applications," *Computer Networks*, vol. 51, no. 4, pp. 1072–1094, 2007.

[9] C. Zhang, A. Krishnamurthy, and R. Y. Wang, "Brushwood: Distributed trees in peer-to-peer systems," in *Peer-to-Peer Systems IV*. Heidelberg: Springer Berlin, 2005, pp. 47–57.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2001, pp. 161–172.

[11] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.