SCALABLE RANGE QUERY PROCESSING FOR LARGE-SCALE DISTRIBUTED DATABASE APPLICATIONS*

Maha Abdallah
LIP6, Université Paris 6
8, rue du Capitaine Scott
75015 Paris, France

Maha.Abdallah@lip6.fr

Hung Cuong Le
LIP6, Université Paris 6
8, rue du Capitaine Scott
75015 Paris, France

cuonglh@poleia.lip6.fr

ABSTRACT
Peer-to-peer (P2P) systems provide a robust, scalable and decentralized way to share and publish data. Although highly efficient, current P2P index structures based on Distributed Hash Tables (DHTs) provide only *exact match* data lookups. This compromises their use in database applications where more advanced query facilities, such as *range queries*, are a key requirement. In this paper, we give a new P2P indexing structure that supports range searches over shared data while maintaining DHTs logarithmic search time. Our index structure can be seen as an extension of the Chord P2P overlay so that data items are mapped to the Chord address space in an order-preserving way, hence supporting range query executions. Load balancing of skewed data is then achieved deterministically using the underlying DHT infrastructure. Experimental evaluations show that our mechanism provides strong load-balancing guarantees in systems with high data skews.

1. Introduction

Peer-to-peer (P2P) computing is emerging as a key paradigm for structuring large-scale distributed systems. P2P systems make it possible to harness resources distributed world-wide in a cost-effective manner by having all nodes participating in the system (called *peers*) play an equal role and act as both clients and servers. The key advantages of P2P systems are scalability, fault-tolerance, absence of central control, and self- organization, allowing peers to join and leave the system at any time.

A fundamental issue in P2P data sharing systems is efficient location of data, and systems have been classified into *unstructured* and *structured* systems with respect to the search technique they employ. In unstructured systems, such as Napster [14] and Gnutella [8], nodes have no knowledge about data items that other nodes maintain. In Napster for instance, a global index of all items available in the system is maintained at a central server. To locate a data item, a node sends a request to the index server and gets the IP address of the node storing the requested item. Subsequent downloads are performed directly between nodes. This centralized approach to locating data introduces a scalability problem in the index server and makes it a single point of failure. To avoid these problems, another extreme approach is followed by Gnutella in which no indexing mechanism is supported. Search is then achieved by flooding requests to all neighboring nodes within a certain scope. Flooding on every node's request clearly does not scale in terms of communication cost and worse, might not find a data item that is actually in the system due to scope limits.

These problems have motivated the research community to focus on the design of structured P2P systems, such as CAN [16], Chord [17], D2B [6], Tapestry [18], and Pastry [4]. In structured systems, a global index of all available data items is maintained and carefully distributed among all participating nodes: every data item is associated with a unique key, and every node in the system is responsible of a set of keys; based on its key, an item (or more precisely, the IP address of the node storing the item) is routed to and stored at the node responsible of that key; key-based queries are then directly routed to that same node. Simply stated, structured P2P systems implement a hash function that maps any given item to a specific node in the system, thereby providing a hash-table-like functionality over multiple distributed nodes; this is why they are commonly referred to as Distributed Hash Tables (DHTs).

The high scalability of DHT systems is due to the scalability of the mechanism by which a key-based request is routed to the node in possession of the requested key, and that without global knowledge of the mapping between keys and nodes. This is achieved by having every node keep information about only a set of neighboring nodes. Efficient routing is then performed by

* This work has been partially funded by the "SemWeb" project of the ACI Masses de Données program.

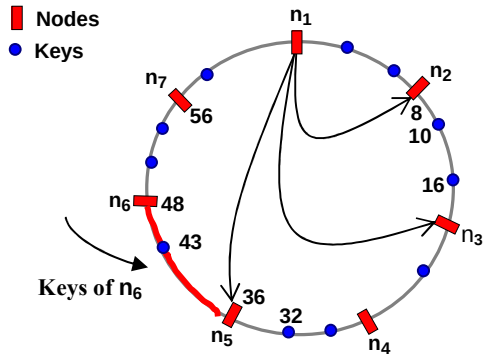

Figure 1. A Chord identifier space

forwarding the request to the neighbor who is "closest" to the destination node. In most DHT systems, this routing is achieved in only $O(\log N)$ overlay hops in a system with N participating nodes.

1.1 Range Queries Vs Load-balancing

A critical issue in DHT systems is load balancing, i.e., the even distribution of items among the nodes in the system. This is usually achieved using a uniform hash function that randomly maps items and nodes to a common *key space*. This removes data skew by ensuring a uniform distribution of items over the key space, and achieves load balancing by having every node be responsible of a balanced portion of the common space.

Although hashing schemes have proved very efficient for load balancing, they unfortunately introduce a serious problem from a database application's viewpoint. Indeed, hashing destroys the order of data items (or any semantic relationship between the items), making any semantic processing over data, such as range queries, impossible. This raises an important question on whether DHTs can be effectively exploited in large-scale database applications where range searches are crucial.

1.2 Our Contribution

In this paper, we answer the above question positively and propose a novel indexing mechanism that builds on the DHT technology to provide high scalability, while enabling efficient execution of range queries in addition to exact-match queries. More precisely, our technique refines the Chord overlay by mapping items to the DHT key space in an order-preserving way, i.e., without hashing. Load balancing is then achieved by moving nodes around the key space depending on the systems' load and data skew so that every node has no more than its fare share from the system.

The rest of the paper is organized as follows. Section 2 defines the underlying system model and gives some background on Chord. Section 3 describes our order-preserving data indexing mechanism, as well as its use to execute range searches. Section 4 validates our proposition and presents simulation results of its performance. Section 5 discusses related works. Finally, section 6 concludes this work and points out some of our future perspectives.

2. System Model

As stated earlier, our indexing mechanism can be seen as a refinement of the Chord DHT so that range queries can be supported while maintaining all the good properties of Chord, namely scalability and high efficiency in dynamic settings. In the following, we will use the terms "node" and "peer" interchangeably.

In Chord, scalability is achieved by avoiding the need that every node knows about every other node in the system: a node needs to maintain routing information about only $O(\log N)$ other nodes, called neighbors. A lookup is then resolved in only $O(\log N)$ messages, where a request is passed from one neighbor to another until the destination node managing the requested item is reached.

To achieve load balancing, Chord uses *consistent hashing* [10] to uniformly distribute items to nodes. More precisely, nodes and items are randomly mapped to an m-bit key space $\mathcal{K} = \{0, \ldots, 2^m - 1\}$ using SHA-1 [5] as a base hash function. A node is mapped to the key space by hashing its IP address, while a data item is mapped to the same key space by hashing the item itself. The image of a node under the hash function is also called its *identifier*. Keys are ordered in a ring, and a data item is assigned to the first node whose identifier is equal to or follows the data item's key in the key space. A *successor* of a node n is defined as the first node that succeeds n on the key ring. Similarly, the *predecessor* of a node n is the first node that precedes n on the key ring. In this setting, the set of neighbors of a node n contains all nodes managing keys succeeding n by $2^i - 1$ in the key space, where $1 \leq i \leq m$. Figure 1 shows a Chord ring with $m = 6$. The system is composed of 7 nodes storing 11 keys. n_6 is the successor node of n_5, while n_5 is the predecessor of n_6. Node n_1 has three neighbors, n_2, n_3, and n_5, that it uses to route lookups to destination nodes.

The major benefit of consistent hashing in Chord, i.e., load-balancing, comes however at a price since hashing scrambles the items over the key space, thereby destroying items' order and making any range search impossible. To deal with this problem, we refine the Chord DHT so as to meet the two antagonistic requirements of load-balancing and range searches. More precisely, we propose to replace Chord's item hashing by any order-preserving mechanism to map items to the key space. However, this might lead to load imbalance in the presence of data skew, where some nodes would have significantly more items landing in their ring interval than other nodes. We solve this problem by extending the Chord infrastructure with a novel mechanism that achieves load-balancing by dynamically moving nodes around the ring depending on the key distribution, which

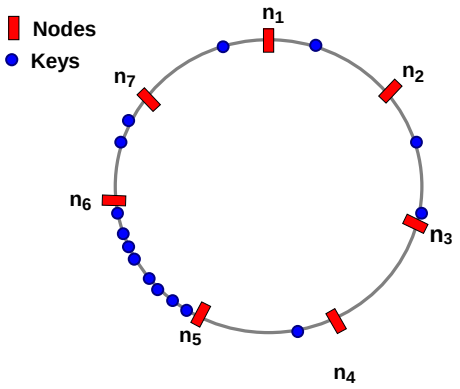

Figure 2. Chord's refinement: order-preserving mapping of items to identifier space

in our case can be imbalanced. Note that apart from item hashing, all the underlying Chord components remain unchanged and can be directly exploited for our purposes, thus inheriting Chord's scalability and routing efficiency.

A node's load is generally defined in terms of the number of keys the node stores, or in terms of the percentage of network bandwidth the node is using. In this paper, we consider the former to be our criterion. Let N be the number of nodes in the system, and K be the number of items stored in the system (where $K >> N$). $L = \lceil K/N \rceil$ clearly refers to the average desired load on nodes in a perfectly balanced system.

Let ℓ_i denote the load of a node n_i. Our objective is to devise a load balancing algorithm that guarantees that n_i's load imbalance, defined as the ratio between ℓ_i and L does not exceed ε, for some constant $\varepsilon \geq 1$. This actually means that $\ell_i \leq \varepsilon L$ should hold for all nodes in the system. A node n_i is said to be *underloaded* whenever ℓ_i drops below the threshold L, and *overloaded* whenever ℓ_i crosses the threshold εL. In this setting, the presence of overloaded nodes intuitively implies the presence of underloaded nodes. The critical issue is then to rebalance the load while preserving initial items' order (i.e., without randomization).

Given that the thresholds on a node's load are defined in terms of the global average load L, it is worth mentioning that the value of L can be efficiently estimated using *epidemic-style proactive aggregation* [12, 9, 13]. These works propose several protocols that enable a network of nodes to collectively compute aggregates, namely averages, in such a way that all nodes know the value of the aggregate continuously and in an adaptive fashion. The resulting protocols are efficient and converge to the true answer exponentially fast.

3. Order-preserving P2P Indexing

As stated earlier, our P2P indexing mechanism refines Chord's infrastructure so as to support range queries while maintaining Chord's scalability and routing efficiency.

3.1 Item Mapping

To meet database applications' requirements in terms of ordered searches over data items, uniform hashing can no longer be exploited to remove data skew. We thus replace item hashing used in Chord by any order-preserving technique ensuring a non-random mapping of items to nodes. For the sake of simplicity, and without loss of generality, we assume that the data items to be indexed are defined over the domain $\{0,\ldots, 2^m - 1\}$. This means that a data item's key/image in the key space is the item itself. Apart from uniform hashing of items, all other Chord components remain unchanged and can be directly used for key search and message routing.

To illustrate, consider the key ring of Figure 1. In our context, node n_6 stores the item whose value is 43, while in Chord, node n_6 would store the item whose image under the hash function is 43. However, as discussed before, using our direct, non-random mapping of items might create load imbalance. Indeed, in the presence of data skew, items would be unevenly distributed over the key space, and some nodes would have much more items assigned to the ring interval they are responsible of, as illustrated in Figure 2. In the following, we propose a novel technique that balances the system's load by moving underloaded nodes to ring intervals owned by overloaded nodes.

3.2 Load-balancing

Similarly to Chord, a node joins the system by hashing its IP address to determine its position on the key ring. As stated earlier, the basic idea underlying our load-balancing mechanism is to have underloaded nodes migrate to zones in the key space where too many data items reside. Our mechanism is deterministic and works as follows. Every node n_i regularly checks its current load ℓ_i. There are three cases to consider:

Case 1: $L \leq \ell_i \leq \varepsilon L$, in which case n_i does nothing.

Case 2: $\ell_i > \varepsilon L$, in which case n_i tries to shed part of its load. Let n_j denote the least loaded of n_i's successor node, n_{i+1}, and n_i's predecessor node, n_{i-1}. There are two subcases to consider:

a) If $(\ell_i + \ell_j)/2 \leq \varepsilon L$, then n_i and n_j start a load balancing procedure. If $n_j = n_{i-1}$, then n_j moves clockwise around the ring by increasing its identifier and takes over $(\ell_i - \ell_j)/2$ items from n_i's interval. This leads to both nodes having $(\ell_i + \ell_j)/2$ items. Figure 3 illustrates this process, with $\varepsilon = 2$ and $L = 2$. Node n_5 has a load $\ell_5 > \varepsilon L$ ($\ell_5 = 6$). Given that $(\ell_4 + \ell_5)/2 \leq \varepsilon L$, n_4 takes over two of n_5's items by increasing its identifier. If, on the other hand, $n_j = n_{i+1}$, then n_i moves itself counterclockwise around the ring by decreasing its identifier, and sheds $(\ell_i - \ell_j)/2$ items to its successor node. Both nodes end up having $(\ell_i + \ell_j)/2$ items.

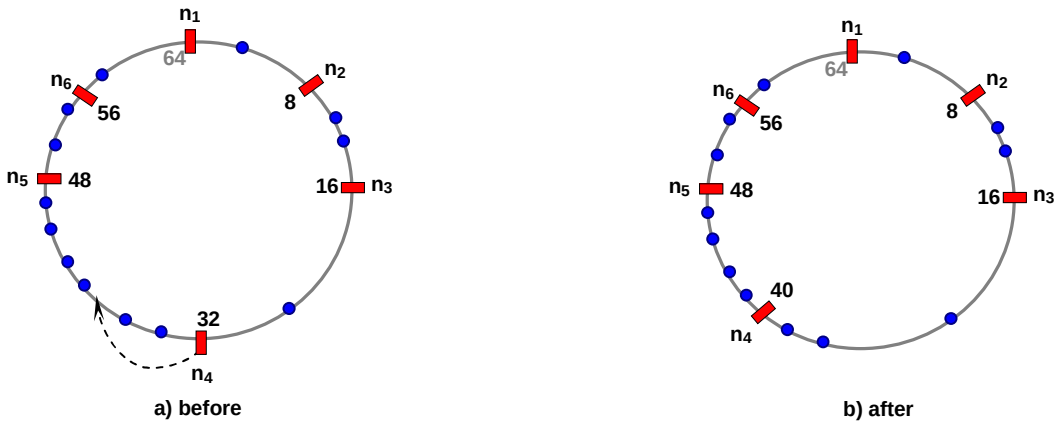

Figure 3. Sharing load with a predecessor node

b) If $(\ell_i+\ell_j)/2 > \varepsilon L$, then n_i contacts a node, say n_k, that has announced itself as underloaded and asks it to share its load. For now, we assume that underloaded nodes are known to other nodes. In section 3.3, we discuss an efficient and scalable technique by which underloaded nodes can be discovered. On n_i's request, n_k leaves the system by handing its own items to its successor node, and rejoins the system by positioning itself on the ring such that it takes in charge half of n_i's load. In other words, n_k's new identifier lies in the interval ring of node n_i so that half of n_i's load is handed to n_k. Figure 4 illustrates this process, with $\varepsilon = 2$ and $L = 2$. Node n_2 has no items in its ring interval, and therefore is underloaded. Node n_5 has a load $\ell_5 > \varepsilon L$ ($\ell_5 = 6$). After discovering n_2, n_5 pulls n_2 to its interval ring and shares with it half of its load. It is important to note that the migration process of n_2 should not leave its successor node, n_3, overloaded. This issue is explained in case 3.

Case 3: $\ell_i < L$. There are two subcases to consider:

a) $\ell_i + \ell_{i+1} > \varepsilon L$, in which case n_i does nothing. This is very important given that if node n_i migrates to a different zone in the ring, all n_i's items would be handed to its successor node, which become overloaded.

b) $\ell_i + \ell_{i+1} \leq \varepsilon L$, in which case n_i announces itself as underloaded, meaning that it is ready to share load with an overloaded node. In the next section, we detail an efficient and scalable technique by which underloaded nodes can advertise themselves so that they can be discovered by overloaded nodes in a deterministic way.

3.3 Publishing/Discovering Underloaded Nodes

As stated earlier, overloaded nodes need to discover underloaded ones to initiate a load balancing process. One solution to this problem is to have a central node keep track of all underloaded nodes by having every node announces itself to the central node whenever case 3(b) of the previous section applies. Underloaded nodes can then be discovered by sending requests to the central node. Clearly, this approach does not scale as the central node quickly becomes a bottleneck and presents a single point of failure. On the other extreme, overloaded nodes can flood the network with a load-sharing request until an underloaded node is found. Flooding on requests overcomes the need for a dedicated node, but is clearly inefficient in terms of communication cost and hence does not scale.

The solution we propose is a tradeoff between these two extremes. More precisely, we propose to maintain a distributed index of underloaded nodes based on the underlying DHT infrastructure. This gives rise to a deterministic solution in which requests for underloaded nodes are resolved in only $O(\log N)$ messages. Towards this end, underloaded nodes wishing to advertise themselves are classified with respect to their load in the following way: Let ℓ_i be the load of an underloaded node n_i. This means that $\ell_i \in U$, where $U = [0, L[$. We divide U into sub-intervals u_r of size s, with the index r running from 1 to $\lceil L/s \rceil$. The value of s can then be tuned as needed, where on the one extreme $s = L$ and on the other $s = 1$.

Depending on its load ℓ_i, an underloaded node n_i is classified into one of the intervals u_r, where $r = \lceil (L-\ell_i)/s \rceil$. The index r is then exploited to advertise n_i by assigning it to a random node in the system using Chord's base hash function. More precisely, r is mapped to Chord's key space by hashing its value.

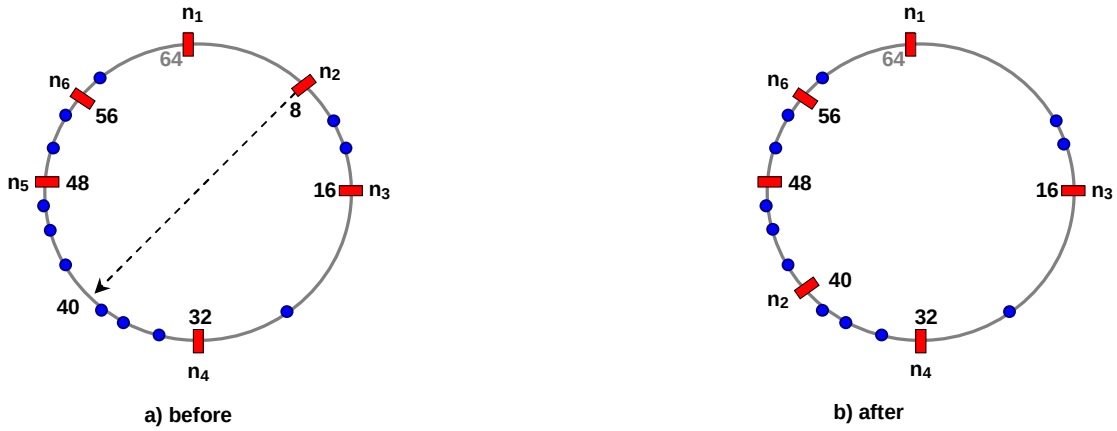

Figure 4. Load balancing by node migration

Based on *r*'s key, an index entry containing the IP address of n_i is routed to and stored at the node responsible of that key; to discover an underloaded node, an overloaded node randomly chooses a value for *r* from the interval [1, $\lceil L/s \rceil$], and maps it to the key space. Key-based queries to locate underloaded nodes are then directly routed in *O*(log *N*) messages to the node responsible of that key. The fact that $K >> N$ ensures a high distribution of the publishing index over the system's nodes.

3.4 Range Query Execution

Given that our protocol maps data items to Chord's identifier space in an order-preserving way, range searches can be simply and efficiently executed based on Chord's infrastructure.

Consider the query for all data items $d \in [v1, v2]$. Using Chord's routing protocol, the first item falling within this range is found in only *O*(log *N*) messages. The query can then be simply forwarded along the ring from one successor node to another until all nodes falling in the query's range are visited. Assuming the query spans *M* nodes, then the query is resolved in only $O(M + \log N)$ messages, thus bringing Chord's scalability and high efficiency to range searches. This makes our protocol an excellent candidate for query execution in highly distributed database applications.

4. Experimental Evaluation

In this section, we present results from the simulation of our load balancing mechanism. At start, the system is highly imbalanced. We set $\varepsilon = 1.5$. The number of overloaded nodes varies in proportion with the network size.

Figure 5 shows the number of overloaded nodes (Y-axis) against the number of load balancing cycles (X-axis) initiated every minute. For this experiment, we conducted simulations on networks ranging from 32 nodes up to 1024 nodes. We observe that our load balancing mechanism brings the system to a balanced state after only three load balancing cycles, where the number of overloaded nodes is drastically reduced. On the seventh cycle, this number reaches zero for all network sizes.

We also studied the effect of dynamism on load balancing. We start with an imbalanced system of 512 nodes. Nodes depart and arrive at random and with random loads.

Figure 6 shows the number of load balancing operations (Y-axis) against the number of cycles (X-axis).We distinguish two operations: (1) load balancing with a successor/predecessor node, and (2) load balancing with a distant node. Indeed, this distinction is important given that the latter operation is less efficient as node migration incurs a more complex reorganization of the network topology. Furthermore, it requires discovering an underloaded node by accessing the associated index structure. We first observe that the number of load balancing operations drops very quickly after only three execution cycles, which reflects the convergence of the system to a stable state observed in the previous experiment after the same number of cycles (cf. Figure 5). With the arrival and departure of nodes, the system's load varies continuously leading to the continuous execution of the load balancing operations, though at a much slower rate. We also observe that load balancing with a successor/predecessor node is executed two to three times more than node migration, thus reducing the cost of overlay maintenance.

Finally, recall that our mechanism inherits its performances from the underlying Chord infrastructure, where message routing cost is logarithmic with the number of nodes in the system.

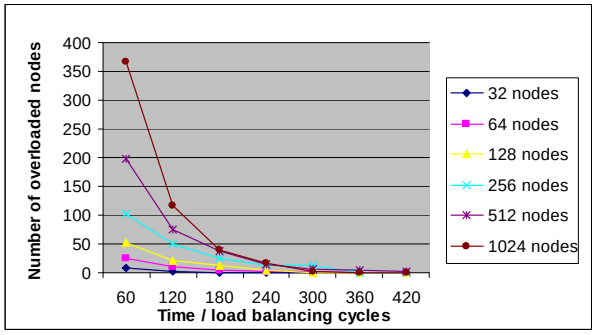

Figure 5. Number of overloaded nodes

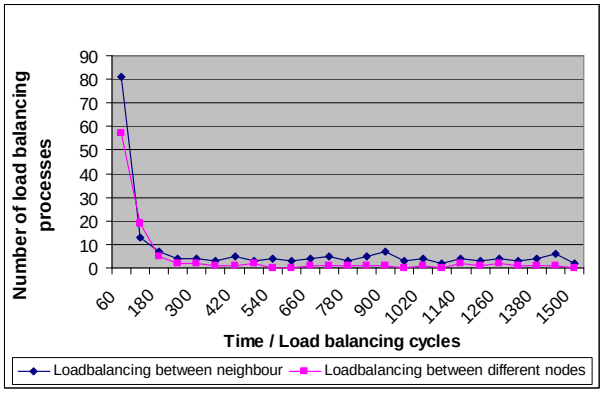

Figure 6. Number of invocations of load balancing operations

5. Related Works

Although P2P computing is emerging as a powerful paradigm for scalable data sharing, work on supporting P2P range queries is still at its beginning.

Recently, distributed range search data structures for P2P systems have been proposed [1, 2, 7, 3]. In [2], the authors describe *skip graph*, a randomized data structure that supports efficient range searches. However, the reported work does not address the issue of assigning items to nodes, and thus do not offer load balance guarantees across nodes. This is why the simplifying assumption of one item per node is made.

[1] extends the work on *skip graphs* by addressing the load balancing problem, however at the price of a highly complex mechanism for adjusting the threshold load dynamically. Furthermore, arriving nodes do not immediately take on load from existing machines, but are instead assigned to a free list, thus affecting the quality of load balancing.

The work reported in [7] describes a load balancing scheme for range partitioned data, and study its use for range searches in P2P systems. The proposed solution depends on building a skip graph on node loads. The major problem with this approach is that skip graphs have a very high maintenance cost that quickly become prohibitive when used over continuously changing data (i.e., node loads) in highly dynamic settings.

[3] describes *p-tree*, a distributed B+-tree-like index structure that provides efficient range searches. However, the proposed structure is based on the simplifying assumption of one data item per node, and does not provide search guarantees in dynamic settings.

As stated in earlier sections, *distributed hash tables* (DHTs) achieve load balancing by applying uniform hashing on keys to remove data skew. This comes at the cost of destroying the ordering of the key space, making any efficient processing of range searches impossible.

Recent works have been conducted to overcome this limitation [11, 15].In [15], the authors address this issue and propose building an additional index over data items on top of DHT's. The resulting structure, called *PHT*, organizes data items into a binary trie built on top of any DHT. The DHT is used to uniformly distribute trie nodes across the system's nodes, thus achieving load balancing. However, the structure suffers from logical data fragmentation, as well as hot spots on the top-level nodes of the trie.

In [11], the authors describe a randomized load balancing algorithm in which lightly loaded nodes move themselves to zones in the key space occupied by too many items, thus taking on load from heavily loaded nodes found by sampling. Our work can be seen as a deterministic variant of [11], which exploits the underlying DHT components to construct a scalable and efficient solution by which lightly loaded nodes can be found deterministically.

6. Conclusions and Future Work

We have presented a distributed data indexing mechanism that supports the efficient execution of range searches over massively distributed databases.

Our mechanism builds on recent P2P technological advances to provide high scalability and efficiency. Our technique refines the DHT technology by using an order-preserving mapping of items to the DHT key space. This enables the efficient execution of range queries in addition to exact-match queries. Load balancing is then achieved in a deterministic way by moving nodes around the DHT key space depending on data skew. An important feature of our technique is that it entirely relies on the underlying DHT infrastructure, thus inheriting its scalability and routing efficiency.

While we have assumed a system in which all nodes have the same capacity, it would be interesting to consider a more heterogeneous setting where some nodes have

much higher capacity than others. One possibility towards this objective is to define a machine-specific load threshold εL, where ε is tuned such that high-capacity nodes can hold more items than others. The same reasoning applies on data items, where it would be interesting to consider data with heterogeneous associated loads by taking into account data item's popularity, storage requirements, and other factors.

References

[1] J. Aspnes, J. Kirsch, and A. Krishnamurthy. Load Balancing and Locality in Range-Queriable Data Structures. In *Proc. of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, July 2004.

[2] J. Aspnes and G. Shah. Skip graphs. In *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2003.

[3] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. Querying Peer-to-Peer Networks Using P-Trees. In *Proc. of the 7th International Workshop on the Web and Databases (WebDB)*, June 2004.

[4] P. Druschel and A. Rowstron. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[5] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, April 1995.

[6] P. Fraigniaud and P. Gauron. Brief announcement: An Overview of the Content-Addressable Network D2B. In *Proc. of the 22nd ACM Symposium on Principles of Distributed Computing (PODC)*, July 2003.

[7] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB)*, August 2004.

[8] Gnutella. http://gnutella.wego.com/.

[9] M. Jelasity and A. Montresor. Epidemic-Style Proactive Aggregation in Large Overlay Networks. In *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, March 2004.

[10] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of the 29th Annual ACM Symposium on Theory of Computing,* May 1997.

[11] D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *Proc. of the 16th ACM Symposium on Parallelism in Algorithms and Architectures(SPAA)*, June 2004.

[12] D. Kempe, A. Dobra, and J. Gehrke. Computing aggregate information using gossip. In *Proc. of the IEEE Symposium on Foundations of Computer Science,* October 2003.

[13] A. Montresor, M. Jelasity, and Ö. Babaoglu. Robust Aggregation Protocols for Large-Scale Overlay Networks. In *Proc. of the IEEE International Conference on Dependable Systems and Networks (DSN)*, June 2004.

[14] Napster. http://www.napster.com/.

[15] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, S. Shenker. Brief announcement: Prefix Hash Tree. In *Proc. of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, July 2004.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of the ACM SIGCOMM 2001 Conference*, August 2001.

[17] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of the ACM SIGCOMM 2001Conference*, August 2001.

[18] Y. B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. *Technical Report UCB/CSD-01-1141, University of California at Berkeley*, 2001.