Replication, Load Balancing and Efficient Range Query Processing in DHTs

Theoni Pitoura[1], Nikos Ntarmos[1], Peter Triantafillou[1]

[1]Research Academic Computer Technology Institute, and Computer Engineering and Informatics Department, University of Patras, Greece
{pitoura, ntarmos, peter}@ceid.upatras.gr

Abstract. We consider the conflicting problems of ensuring data-access load balancing and efficiently processing range queries on peer-to-peer data networks maintained over Distributed Hash Tables (DHTs). Placing consecutive data values in neighboring peers is frequently used in DHTs since it accelerates range query processing. However, such a placement is highly susceptible to load imbalances, which are preferably handled by replicating data (since replication also introduces fault tolerance benefits). In this paper, we present HotRoD, a DHT-based architecture that deals effectively with this combined problem through the use of a novel locality-preserving hash function, and a tunable data replication mechanism which allows trading off replication costs for fair load distribution. Our detailed experimentation study shows strong gains in both range query processing efficiency and data-access load balancing, with low replication overhead. To our knowledge, this is the first work that concurrently addresses the two conflicting problems using data replication.

1 Introduction

Structured peer-to-peer (P2P) systems have provided the P2P community with efficient and combined routing and location primitives. This goal is accomplished by maintaining a structure in the system, emerging by the way that peers define their neighbors. Different structures have been proposed, the most popular of which being: distributed hash tables (DHTs), such as CAN [17], Pastry [18], Chord [21], Tapestry [24], which use hashing schemes to map peers and data keys to a single, modular identifier space; distributed balanced trees, where data are stored at the nodes of a tree, such as P-Grid[1], PHT [16], BATON [11], etc.

One of the biggest shortcomings of DHTs that has spurred considerable research is that they only support exact-match queries. Therefore, the naïve approach to deal with range queries over DHTs would be to individually query each value in the range, which is greatly inefficient and thus infeasible in most cases. Although there are many research papers that claim to support range queries over DHTs more "cleverly" and, thus, efficiently ([2], [9], [19], [22]), all of them suffer from access load imbalances in the presence of skewed data-access distributions. Only a few approaches deal with both problems, i.e. load balancing and efficient range query processing, in DHTs ([5]), or other structures ([3], [7], [11]). However, these solutions are based on data migration which is sometimes inadequate in skewed data access distributions. This is

more apparent in the case of a single popular data value which makes the peer that stores it heavily loaded. Transferring this value to another peer only transfers the problem. In such cases, access load balancing is best addressed using replication of popular values to distribute the access load among the peers storing such replicas.

In this work we propose solutions for efficiently supporting range queries together with providing a fair load distribution over DHTs using replication. Our approach is based on two key ideas. The first is to use locality-preserving data placement, i.e. to have consecutive values stored on neighboring peers; thus, collecting the values in a queried range can be achieved by single-hop neighbor to neighbor visits. The second is to replicate popular values or/and ranges to fairly distribute access load among peers. However, using data replication together with a locality-preserving data placement is not simple: if the replicas of a popular value are placed in neighboring peers, the access load balancing problem still exists in this neighborhood of peers that is already overloaded; On the other hand, if the replicas are randomly distributed, additional hops are required each time a replica is accessed during range query processing. Addressing these two conflicting goals is the focus of this paper.

Specifically, we make the following contributions:

1. We define a novel locality-preserving hash function, used for data placement in a DHT, which both preserves the order of values and handles value/range replication. The above can be applied to any DHT with slight modifications (we use Chord [21] for our examples and in our experiments).
2. We propose a tunable replication scheme: by tweaking the degree of replication, a system parameter, we can trade off replication cost for access load balancing. This is useful when we know, or can predict the characteristics of the query workload.
3. We develop a locality-preserving, DHT architecture, which we coin HotRoD, that incorporates the above contributions, employing locality-preserving replication to ensure access-load balancing, and efficient range query processing.
4. We comprehensively evaluate HotRoD. We propose the use of a novel load balancing metric, Lorenz curves and the Gini coefficient (which is being heavily used in other disciplines, such as economics and ecology), that naturally captures the fairness of the load distribution. We compare HotRoD against baseline competitors for both range query processing efficiency and load distribution fairness. Further, we study the trade-offs in replication costs vs. achievable load balancing.
5. Our results from extensive experimentation with HotRoD show that HotRoD achieves its main goals: significant speedups in range query processing and distributes accesses fairly to DHT nodes, while requiring only small replication overhead. Specifically, a significant hop count saving in range query processing, from 5% to 80% compared against standard DHTs. Furthermore, data-access load is significantly more fairly distributed among peers, with only a small number of replicas (i.e. less than 100% in total). As the range query spans, or data-access skewness increases, the benefits of our solution increase.

To our knowledge, this is the first work to concurrently address the issues of *replication-based data-access load balancing* and *efficient range query processing* in structured P2P networks and study in detail its performance features.

The rest of the paper is organized as follows: In section 2 we introduce the HotRoD architecture, its locality-preserving hash function, and the mechanisms for replica management, and in section 3 we present the algorithm for range query process-

ing. In section 4 we experimentally evaluate HotRoD, and present its ability to tune replication. Finally, we discuss related work, in section 5, and conclude in section 6.

2 HotRoD: A Locality-Preserving Load Balancing Architecture

The main idea behind the proposed architecture is a novel hash function which: (a) preserves the ordering of data to ensure efficient range query processing, and, (b) replicates and fairly distributes popular data and their replicas among peers.

HotRoD is built over a locality-preserving DHT, i.e. data are placed in range partitions over the identifier space in an order-preserving way. Many DHT-based data networks are locality-preserving (Mercury [5], OP-Chord [22, 15], etc) in order to support range queries. However, this additional capability comes at a price: locality-preserving data placement causes load imbalances, whereas trying to provide load balancing, the order of data breaks. HotRoD strives for a uniform access load distribution by replicating popular data across peers in the network: its algorithms detect overloaded peers and distribute their access load among other, underloaded, peers in the system, through replication. (We should mention that instances of the algorithms run at each peer, and no global schema knowledge is required).

In the following sub-sections, we overview the underlying locality-preserving DHT, define a novel locality-preserving hash function, and algorithms to detect load imbalances and handle data replication and load balancing.

2.1 The Underlying Locality-Preserving DHT

We assume that data objects are the database tuples of a k-attribute relation $R(A_1, A_2, \ldots, A_k)$, where A_i $(1 \le i \le k)$ are R's attributes. The attributes A_i are used as single-attribute indices of any tuple t in R. Their domain is DA_i, for any $1 \le i \le k$. Every tuple t in R is uniquely identified by a primary key, *key(t)*, which can be either one of its A_i attributes, or calculated by more than one A_i attributes.

In DHT-based networks, peers and data are assigned unique identifiers in an m-bit identifier space (here, we assume an identifier ring modulo-2^m). Traditional DHTs use secure hash functions to randomly and uniquely assign identifiers to peers and data. Here, a tuple's identifier is produced by hashing its attributes' *values* using k (at most[1]) order-preserving hash functions, $hash_i()$, to place tuples in range partitions over the identifier space. For fault tolerance reasons, a tuple is also stored at the peer mapped by securely hashing its *key(t)*. Thus, data placement requires $O((k+1)\cdot logN)$ hops - N is the number of peers ([22]).

Note: We may apply an additional level of indirection by storing pointers to tuples, as index tuples $I_i(t)$*:* $\{v_i(t)$ key$(t)\}$*, instead of tuples themselves. At this point, we make no distinction.*

[1] Functions $hash_i()$ may be different for each one of the k different attributes A_i.

As most existing DHTs, tuples use consistent hashing ([12]): a tuple with identifier *id* is stored at the peer whose identifier is the "closest" to *id* in the identifier space (i.e. the successor function, *succ*(), of Chord [21]). Peers also maintain routing information about peers that lie on the ring at logarithmically increasing distance (i.e. the *finger tables* of Chord [21]). Using this information, routing a message from one peer to another requires O(*logN*) hops in the worst case, where *N* is the number of peers. For fault-tolerance reasons, each peer also maintains a maximum of *logN* successors.

Example 2.1. Fig 1 illustrates data placement in a 14-bit order-preserving Chord-like ring, i.e. the id space is [0, 16383]. We assume single-attribute A tuples, DA=[0, 4096). Let N=7 peers inserted in the network with identifiers 0, 2416, 4912, 7640, 10600, 11448, and 14720. Each peer is responsible for storing a partition of the attribute domain DA, in an order-preserving way, as shown.

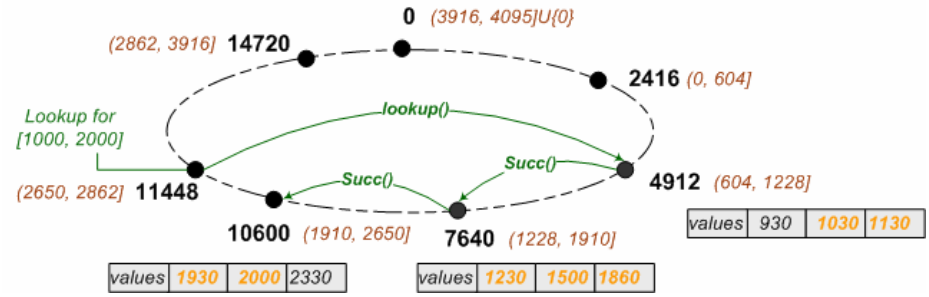

Fig 1. The substrate locality-preserving DHT. (a) A tuple with value $v \in DA$ is stored on peer *succ(hash(v))*, (b) The range query [1000, 2000] is routed from peer *succ(hash(1000))*=4912, through the immediate successors, to peer *succ(hash(2000))*=10600

A range query is pipelined through those peers whose range of index entries stored at them overlaps with the query range. It needs O(logN+ n') routing hops – n' is the number of these peers ([22]).

Example 2.2. Fig 1 also illustrates how the range query [1000, 2000] initiated at peer 11448 is answered. Using the underlying DHT network look up operation, lookup() (i.e. Chord lookup, if the underlying network is Chord), we move to peer succ(hash(1000)), which is peer 4912. Peer 4912 retrieves all tuples whose values fall into the requested range, and forwards the query to its successor, peer 7640. The process is repeated until the query reaches peer 10600 (i.e. succ(hash(2000))), which is the last peer keeping the requested tuples.

Although it accelerates routing for range queries, this scheme cannot handle load balancing in the case of skewed data-access distributions. HotRoD, the main contribution of this work, deals with this problem while still attaining the efficiency of range query processing. From this point forward, we assume that R is a single-attribute index A relation, whose domain is *DA*. Handling multi-index attribute relations is straightforward ([14]), and beyond the scope of this paper.

2.2 Replication and Rotation

Each peer keeps track of the number of times, α, it was accessed during a time interval, and the average low and high bounds of the ranges of the queries it processed, at this time interval – *avgLow* and *avgHigh* respectively. We say that a peer is overloaded, or "*hot*" when its access count exceeds the upper limit of its resource capacity, i.e. when $\alpha > \alpha_{max}$. An arc of peers (i.e. successive peers on the ring) is "*hot*" when at least one of these peers is hot.

In our scheme, "hot" arcs of peers are replicated and rotated over the identifier space. Thus, the identifier space can now be visualized as a number of replicated, rotated, and overlapping rings, the Hot Ranges/Rings of Data, which we call HotRoD (see fig 2). A HotRoD instance consists of a regular DHT ring and a number of virtual rings where values are addressed using a multi-rotation hash function, *mrhf()*, defined in the following sub-section. By the term "virtual" we mean that these rings materialize only through *mrhf()*; there are no additional successors, predecessors or other links among the peers in the different rings.

2.3 Multi-Rotation Hashing

We assume that $\rho_{max}(A)$ is the maximum number of instances that each value of an attribute A can have (including the original value and its replicas). This parameter depends on the capacity of the system and the access load distribution of A's values; indicative values for $\rho_{max}(A)$ are discussed in section 4.4. We also define the index variable $\delta \in [1, \rho_{max}(A)]$ to distinguish the different instances of A's values, i.e. an original value v corresponds to δ=1 (it is the 1st instance of v), the first replica of v corresponds to δ=2 (it is the 2nd instance of v), and so on. Then, the δ^{th} instance of a value v is assigned an identifier according to the following function, *mrhf()*[2].

Definition 1: mrhf(). For every value, $v \in DA$, and $\delta \in [1, \rho_{\max}(A)]$, the Multi-Rotation Hash Function (MRHF) $mrhf : DA \times [1, \rho_{\max}(A)] \rightarrow \{0,1,\ldots,2^m - 1\}$ is defined as:

$$mrhf(v, \delta) = (hash(v) + random[\delta] \cdot s) \bmod 2^m \quad (1)$$

where $s = 1/\rho_{\max}(A) \cdot 2^m$ is the rotation unit (or else, "stride"), and *random*[] is a pseudo-random permutation of the integers in $[1, \rho_{\max}(A)]$ and *random*[1]=0.

It is obvious that for δ=1, *mrhf()* is a one-to-one mapping from *DA* to {0, 1, …, 2^m-1} and a mod2^m order-preserving hash function. This means that, if v and $v' \in$ DA and $v \le v'$, then $mrhf(v,1) \le_{\text{mod}2^m} mrhf(v',1)$, which means that $mrhf(v,1)$ lies before $mrhf(v',1)$ in a clockwise direction over the identifier ring. For any δ>1, HotRoD is

[2] Mapping data to peers (i.e. using consistent hashing) are handled by the underlying DHT.

also mod2^m order-preserving (the proof is straightforward and omitted for space reasons).

Therefore, a value v is placed on the peer whose identifier is closer to $mrhf(v,1)$, according to the underlying DHT. When the δ^{th} instance of a value v is created, or else the $(\delta\text{-}1)^{th}$ replica of v (i.e. $\delta>1$), it will be placed on the peer whose identifier is closer to $mrhf(v,1)$ shifted by $\delta \cdot s$ clockwise. This can be illustrated as a clockwise rotation of the identifier ring by $\delta \cdot s$, and, thus, s is called rotation unit, whereas δ is also referred as the number of rotations.

Example 2.3. Fig 2 illustrates a HotRoD network with δ=2, where ring 1 is the network of fig 1. We assume that peers 4912 and 7640 are "hot", and, thus, have created replicas of their tuples in the peers 14720, and 0. Let s=8191 (i.e. half the identifier space). The partitions of the attribute domain that these peers are responsible to store in the ring 2 (i.e. the first replicas) are shown in the figure.

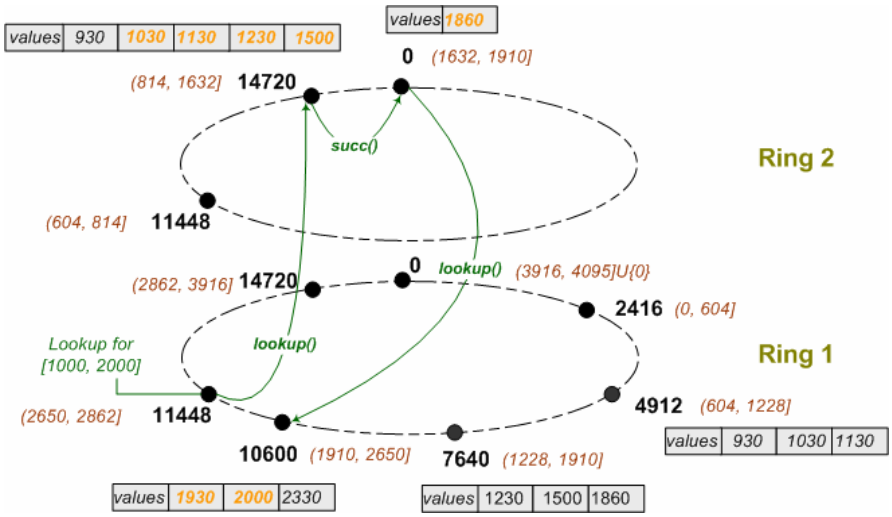

Fig 2. HotRoD for δ=2. (a) The hot peers 4912, 7640 create replicas of their tuples at peers 14720, 0 of ring 2. (b) The range query [1000, 2000] initiated at peer 11448 is sent to peer mrhf(1000, 2)=14720 at ring 2, then to 0, and it jumps to ring 1, to complete

mrhf() leverages the existence of a maximum of $\rho_{max}(A)$ replicas per value v, thus being able to choose one out of $\rho_{max}(A)$ possible positions for v in the system. That way it fights back the effects of load imbalances caused by *hash()* (which are explained in [22]). Note that *randomly* selecting replicas, using *random*[], leads to a uniform load distribution among replica holders. The result can be thought of as superimposing multiple rotated identical rings (as far as data is concerned) on each other, and projecting them to the original unit ring. Thus, "hot" (overloaded) and "cold" (underloaded) areas of the rings are combined through rotation, to give a uniform overall "temperature" across all peers.

2.4 Replicating Arcs of Peers: Implementation Issues

We assume that each peer keeps $\rho_{max}(A)$, the per-attribute maximum number of instances of a value of attribute A (and, thus, it can calculate stride s)[3]. In addition, each peer can calculate the highest value of DA that it is responsible to store at a specific ring; we call it *higherValue* (this is achieved through the reverse function of *mrhf()*, $mrhf^{-1}()$).

We also define $\rho(v(A))$ to be the replication factor for a value $v(A)$, i.e. the (current) number of its replicas, which should be equal to, or less than $\rho_{max}(A)$. Each peer must have "write" access to this measure during replication (see PutRho() below), or "read" access during query processing and data placement (see GetRho() below).

When a peer, p, is detected "hot", it starts replication. Instead of replicating a single peer, we decide to replicate arcs of peers, and specifically the arc consisting of p's successive neighbors that correspond to the range [avgLow, avgHigh]. In that way, costly jumps between rings during range query processing are reduced; jumps between rings happen when different replication factors exist between consecutive values (i.e. when two *non successive peers* store their replicas in one peer at a higher ring, whereas the peers that lie between them in the lower ring do not have replicas at the higher ring).

In terms of simplicity, in the algorithms presented below we assume that replication is only initiated at the original ring, i.e. ring 1.

Each peer periodically (or upon a request of another peer) runs the algorithm REPLICATE_ARC() which detects whether it is hot, or not (if $\alpha > a_{max}$); if it is hot, it creates replicas of its tuples, and sends replication messages, CREATE_REPLICA(), to both its successor (*succ()*) and predecessor (*pred()*). The number of replicas that creates is equal to $rho = \max(\lceil \alpha / \alpha_{\max} \rceil, \{\rho(v(A)),\ v(A) \in [avgLow, avgHigh]\})$ (if $rho \leq \rho_{max(A)}$). Upon receiving a replication message, a peer creates *rho* replicas of those tuples that have less than *rho* replicas, and sends replication messages to its successor (or predecessor, depending on which peer sent the message). Besides, each peer sets the replication factor $\rho(v(A))$ equal to *rho,* for all values $v(A) \in$ [avgLow, avgHigh] that had less than *rho* replicas. The message is sent to all peers that are responsible to store all values $v \in [avgLow, avgHigh]$, which form an arc on the identifier ring.

The pseudocode follows (it uses the inverse function of MRHF, $mrhf^{-1}$).

```
1. REPLICATE_ARC()
2.   /* p is the current peer */
3. BEGIN
4.  rho = ceiling(a / a_max);
5.  if (rho <= 1) exit;
6.  for each v(A) , v(A) >= avgLow and v(A) <= avgHigh {
```

[3] We assume integer domains, whereas real domains can be handled in a similar way. Attribute domains other than integer/real valued can be handled by converting them to an appropriate integer/real form. Note that this conversion is also central to the design of range queries; e.g. range queries for string-valued attributes ought to define some sort of binary comparison operator between values of the attribute.

```
7.      tmp = GetRho(v(A));
8.      rho = max(rho, tmp); }
9.  if (rho > ρmax(A))  rho = ρmax(A);
10. for each tuple t in p, and v(A) ∈ t
11.     copy t to succ(mrhf(v(A), k)), for all k:ρ(v(A)) ≤ k ≤ rho;
12. for each value v(A) in (avgLow, avgHigh) {
13.     if (ρ(v(A)) < rho) putRho(v(A), rho); }
14. send create_replica(p,(avgLow, avgHigh), rho, 1) to succ(p);
15. send create_replica(p,(avgLow, avgHigh), rho, 0) to pred(p);
16. END

17. CREATE_REPLICA(n, (low, high), rho, succ)
18.   /* n is the initiator peer; succ is equal to 1/0, if the
        message is  propagated through successsor/predecessor l
        inks; p is the current peer */
19. BEGIN
20.   higherValue = mrhf^-1(p, 1);
21.   if (succ==0 and higherValue<low)
22.     exit;
23.   for each tuple t in p, and v(A) ∈ t
24.     copy t to succ(mrhf(v(A), k)), for all k:ρ(v(A)) ≤ k ≤ rho;
25.   for each value v(A) in (avgLow, avgHigh) {
26.     if (ρ(v(A)) < rho) putRho(v(A), rho); }
27.   if (succ==1 and higherValue<high)
28.     send create_replica(n,(avgLow, avgHigh),rho,1) to succ(p);
29.   else if (succ==0)
30.     send create_replica(n,(avgLow, avgHigh),rho,0) to pred(p);
31. END
```

Functions GetRho(), PutRho() manipulate the replication factor, $\rho(v(A))$ of an attribute value $v(A)$ over the network; the former gets $\rho(v(A))$, while the latter sets $\rho(v(A))$ equal to a specific number. The replication factor, $\rho(v(A))$, is uniformly hashed in the underlying DHT architecture (using the secure hash function). The initial values for $\rho(v(A))$ is 1, for all $v(A) \in DA$. Since both functions use the underlying DHT architecture, their hop-count complexity is O(*logN*).

Please note that we do not necessarily replicate all tuples that belong to a peer which is replicated. We replicate only the tuples whose values have fewer replicas than target *rho* (this concerns only the first and last peer of the arc). This reduces replication costs without affecting the efficiency of range query processing; we simply assume that each peer keeps track of the ranges that stores at each ring it belongs to.

2.5 Fault-tolerance and High Availability

The existence of replicas in addition to being critical for load balancing purposes is instrumental in providing increased data availability and fault-tolerance during query processing. Although details are beyond the scope of this paper, HotRoD can straightforwardly provide fault tolerance as follows: when a peer storing a queried value does

not respond, the requesting peer simply selects another ρ-value and redirects the query to the peer which keeps a replica of the queried values at a different ring. This continues until one available replica is retrieved.

2.6 Managing Tuple Updates

Tuple Insertion. The peer publishing the tuple stored the tuple at peer *succ*(mrhf(v(*A*), 1)), and checks if $\rho(v(A)) > 1$. If true, it creates $\rho(v(A))$-1 replicas of the tuple (or of its indices) and stores them to *succ*(*mrhf*(*v*(*A*), *k*)), for $2 \leq k \leq \rho(v(A))$.

This operation needs O(*logN*) hops to retrieve $\rho(v(A))$ plus $O(\rho_{max}(A) \cdot logN)$ hops when $\rho(v(A)) > 1$, in the worst case (since $\rho(v(A)) \leq \rho_{max}(A)$).

Tuple Deletion. A tuple deletion message is sent to peer *succ*(*mrhf*(*v*(*A*), 1)) and to all $\rho(v(A))$-1 replica holders, if $\rho(v(A)) > 1$. In addition, peer *succ*(*mrhf*(*v*(*A*),1)) checks if there are other tuples having value *v*(*A*), and if not, it sets $\rho(v(A))$ equal to 1.

The cost of a tuple deletion is, in the worst case, $O((\rho_{max}(A)+2)\cdot \log N))$ hops (including a GetRho() operation to get $\rho(v(A))$, and a PutRho() operation, to set $\rho(v(A))$ equal to 1, if needed).

Tuple Update. It consists of one tuple deletion and one tuple insertion operations.

Naturally, as with all data replication strategies, the load balancing and fault tolerance benefits come at the expense of dealing with updates. However, our experimental results (presented below) show that with a relatively small overall number of replicas our central goals can be achieved, indicating that the relevant replication (storage and update overheads) will be kept low.

2.7 Discussion

Optimal ρ_{max} Values

The calculation of optimal $\rho_{max}(A)$ is important for the efficiency and scalability of HotRoD. This value should be selected without assuming any kind of global knowledge. Fortunately, the skewness of expected access distributions has been studied, and it can be given beforehand; for example, the skewness parameter (i.e. theta-value) for the Zipf distribution ([20]). Given this, and the fact that each additional replica created is expected through HotRoD to take on an equal share of the load, our present approach is based on selecting a value for $\rho_{max}(A)$ to bring the total expected hits for the few heaviest-hit peers (e.g., 2-3%) close to the expected average hits all peers would observe, if the access distribution was completely uniform.

Data Hotspots at ρ-value Holders

In order to avoid creating new data hotspots at the peers responsible for storing $\rho(v(A))$ of a value *v*(*A*), our approach is as follows:

- $\rho_{max}(A)$ instances for this metadata information (ρ-value) of each value can be easily maintained, with each replica selected at random at query start time (recall that $\rho_{max}(A)$ is kept in each peer).
- The hottest values are values that participate in a number of range queries of varying span. Thus, all these queries may start at several different points.

3 Range Query Processing

Consider a range query $[v_{low}(A), v_{high}(A)]$ on attribute A initiated at peer p_{init}. A brief description of the algorithm to answer the query follows: peer p_{init} randomly selects a number, r from 1 to $\rho(v_{low}(A))$, the current number of replicas of $v_{low}(A)$. Then, it forwards the query to peer p_l: $succ(mrhf(v_{low}(A), r))$. Peer p_l searches for matching tuples and forwards the query to its successor, p. Peer p repeats similarly as long as it finds replicas of values of R at the current ring. Otherwise, p forwards the range query to a (randomly selected) lower-level ring and repeats. Processing is finished when all values of R have been looked up. The pseudocode of the algorithm follows:

```
1. PROCESS_RANGE_QUERY(p_init, (v_low(A),v_high(A)) )
2. BEGIN
3.   rho = GetRho(v_low(A));
4.   r = random(1, rho);
5.   send Forward_Range(p_init, (v_low(A), v_high(A)), r) to
              succ(mrhf(v_low(A)), r);
6. END

7. FORWARD_RANGE(p_init, (v_l(A), v_h(A)), r)
8.   /* p is the current peer */
9. BEGIN
10.   Search p locally and send matching tuples to p_init;
11.   higherValue = mrhf^-1(p, r);
12.   if (higherValue < v_h(A)) {
13.     v_next(A) = higherValue+1;
14.     rho = GetRho(v_next(A));
15.     if (rho >= r)
16.       send Forward_Range(p_init,(v_next(A),v_h(A)),r) to succ(p);
17.     else {
18.       r_next=random(1, rho);
19.       send Forward_Range(p_init,(v_next(A), v_h(A)), r_next)
               to succ(mrhf(v_next(A), r_next)); } }
20. END
```

higherValue of p is used to forward the query to the peer responsible for the lowest value of DA that is higher than *higherValue*. Let this value be $v_{next}(A)$. If there is such a peer in ring r (i.e. this happens when $\rho(v_{next}(A))$ is equal to, or higher than r), p forwards the query to its successor. Otherwise, p sends the query to a peer at a lower-level ring, selected randomly from 1 to $\rho(v_{next}(A))$ (using the lookup operation of the underlying DHT). This happens when the range consists of values with different

number of replicas. The algorithm finishes when the current *higherValue* is equal to, or higher than $v_{high}(A)$.

Example 3.1. Fig 2b illustrates how the range query of example 2.2 is processed in HotRoD. First, we assume that peer 11448 forwards the query to peer 14720, i.e. lookup(mrhf(1000, 2)). Moving through successors, the query reaches peer 0. But, the range partition (1910, 2000] is not found at ring 2. Therefore, the query "jumps" to ring 1, peer 10600 (i.e. lookup(mrhf(1911, 2))), where it finishes.

4 Experimental Evaluation

We present a simulation-based evaluation of HotRoD. The experiments have been conducted on a heavily modified version of the internet-available *Chord* simulator, extended to support relations, order-preserving hashing, replication, and range queries.

We compare the performance of HotRoD against:

- *Plain Chord (PC),* as implemented by the original Chord simulator;
- an imaginary *enhanced Chord (EC),* assuming that for each range the system knows the identifiers of the peers that store all values of the *range*;
- *OP-Chord*, a locality preserving Chord-based network ([22, 15])

The results are presented in terms of:

a. *efficiency of query processing,* mainly measured by the number of hops per query, assuming that for each peer, the local query processing cost is O(1) ;
b. *access load balancing*, measured by the cumulative access load distribution curves and the Gini coefficient (defined below);
c. *overhead costs*, measured by the number of peers' and tuples' replicas.

4.1 Simulation Model

The experiments are conducted in a system with N =1,000 peers, and a maximum of 10 (i.e. log$N$) finger table entries and 10 immediate successors for each peer. We use a single-index attribute relation over a domain of 10,000 integers, i.e. DA=[0, 10,000).

We report on 5,000 tuples and a series of 20,000 range queries generated as follows: the mid point of a range is selected using a Zipf distribution ([20]) over DA with a skew parameter θ taking values 0.5, 0.8, and 1. The lower and upper bounds of a range are randomly computed using a *maximum* range span equal to $2 \cdot r$, for a given parameter r (i.e. r is equal to the average range span). In our experiments, r ranges from 1 to 400, and, thus, yielding an average selectivity from 0.01% to 4% of the domain size DA.

Finally, we present experimental results of the HotRoD simulator with different maximum numbers of instances, $\rho_{max}(A)$, ranging from 2, i.e. one replicated Chord ring, to 150 (in this section, $\rho_{max}(A)$ is denoted as ρ_{max}), to illustrate the trade-off load imbalances with replication overhead costs. We should mention here that the reported load imbalances are collected when the system has entered a *steady state* with respect to the peer population and the number of replicas.

4.2 Efficiency of Query Processing

Chord and OP-Chord resolve *equality queries* (i.e. r =1) in ½ ·$logN$ hops, on average. In HotRoD, this becomes $logN$ since two Chord lookup operations are needed: one for the GetRho() operation and one for the lookup operation on the selected ring.

Let a *range query RQ* of span r (i.e. there are r integer values in the query). We assume that the requested index tuples are stored on n *peers* under Chord and enhanced Chord, and on n' *peers* under OP-Chord. Thus, the average complexity of the range query processing is estimated as follows:

- *PC*: r equality queries are needed to gather all possible results (one for each one of the values belonging to *RQ*) for an overall hop count of O(r·$logN$).
- *EC*: n equality queries must be executed to gather all possible results for an overall hop count of O(n·$logN$).
- *OP-Chord* and *HotRoD:* one lookup operation is needed to reach the peer holding the lower value of *RQ* ($logN$ hops), and n'*-1* forward operations to the successors (n'*-1 hops*), for a final overall hop count of $O(logN + n')$; note that the constant factor hidden by the big-O notation is higher in HotRoD, due to the GetRho() operations needed to be executed first of all.

The experimental results in terms of hop counts per range query are shown in Table 1. Comparing HotRoD against OP-Chord, we conclude, as expected, that HotRoD is more expensive; the extra hops incurred are due to the GetRho() operations, which facilitate load balancing. We should note, however, that HotRoD compares very well even against EC, ensuring hop-count savings from 4% to 78% for different r's. As r increases, the hop-count benefits of OP-Chord/HotRoD versus PC/EC increase.

Table 1. Average number of hops per query for different range spans r (θ = 0.8)

r	50	100	200	400
PC	123	246	489	898
EC	25	48	87	190
OP–Chord	18	20	25	33
HotRoD (ρ_{max}=30)	24	27	31	41

4.3 Access Load Balancing

We compare load balance characteristics between OP-Chord and HotRoD. We use the access count, α, which, as defined above, measures the number of successful accesses per peer (i.e. *hits*). We illustrate results using the Lorenz curves and the Gini Coefficient, borrowed from economics and ecology because of their distinguished ability to capture the required information naturally, compactly, and adequately.

Lorenz curves ([6]) are functions of the cumulative proportion of ordered individuals mapped onto the corresponding cumulative proportion of their size. In our context, the ordered individuals are the peers ordered by the number of their hits. If all peers have the same load, the curve is a straight diagonal line, called the *line of equality*, or *uniformity* in our context. If there is any imbalance, then the Lorenz curve falls below the line of uniformity. Given n ordered peers with l_i being the load of peer i,

and $l_1 \leq l_2 \leq \ \ldots \leq l_n$, the Lorenz curve is expressed as the polygon joining the points *(h/n, L_h/L_n)*, where *h=0, 1, 2, ..., n,* L_0= 0, and $L_h = \sum_{i=1}^{h} l_i$.

The total amount of load imbalance can be summarized by the Gini coefficient (*G*) ([6]), which is defined as the relative mean difference, i.e. the mean of the difference between every possible pair of peers, divided by their mean load. It is calculated by:

$$G = \sum_{i=1}^{n} (2 \cdot i - n - 1) \cdot l_i \Big/ n^2 \cdot \mu, \qquad (3)$$

where μ is the mean load. *G* also expresses the ratio between the area enclosed by the line of uniformity and the Lorenz curve, and the total triangular area under the line of uniformity. *G* ranges from a minimum value of 0, when all peers have equal load, to a maximum of 1, when every individual, except one has a load of zero. Therefore, as *G* comes closer to 0, load imbalances are reduced, whereas, as G comes closer to 1, load imbalances are increased.

We should mention here that *G*=0 if and only if *all* peers in the network have *equal* load. However, this is extremely rare in a P2P network. Therefore, we measured *G* in different setups with different degrees of fairness in load distributions. We noticed that *G* was very close to 0.5 in all setups with quite a fair load distribution. In general, in fair load distributions *G*'s values ranged from 0.5 to 0.65, whereas in very unfair load distribution, from 0.85 to 0.99. Therefore, our target is to achieve values of *G* close to 0.5. Besides, *G* is used as a summary metric to compare load imbalances between different architectures (i.e. PC, EC, etc) and different setups.

We ran experiments with different range spans, *r*'s, and Zipf parameters, θ's. In figure 3, hits distribution is illustrated for *r*=200 and θ=0.8 (here, HotRoD ran with ρ_{max} = 400, and ρ_{max}=15). The Gini coefficient (*G*) in PC and EC is 0.78[4], in OP-Chord 0.87, and in HotRoD 0.53. *G* in HotRoD is significantly reduced comparing to the other architectures, with a decrease of 32% comparing with PC/EC and 39% comparing to OP-Chord. The results of experiments with lower range spans are similar. As example, for *r*=50 and θ=0.8, *G* in PC and EC is 0.81, in OP-Chord 0.95, whereas in HotRoD (ρ_{max}=100, ρ_{max}=50) *G* is 0.64, i.e. decreased by 20% and 32%, respectively (see figure 4). Both examples show clearly how HotRoD achieves a great improvement in access load balancing. All experiments have shown similar results.

Furthermore, the resulting Lorenz curves (figures 3 and 4) show that the top 3% heaviest-hit peers receive about an order of magnitude fewer hits in HotRoD than in OP-Chord. At the same time, the mostly-hit of the remaining (underutilized) 97% of the peers receive a hit count that is very slightly above the load they would receive if the load was uniformly balanced. The load balancing benefits and key philosophy of HotRoD are evident in Lorenz curves. HotRoD attempts to off-load the mostly-hit peers by involving the remaining least-hit peers. Thus, intuitively, we should expect to see a considerable off-loading for the heaviest-hit peers, while at the same time, we should expect to see an increase in the load of the least-hit peers.

---

[4] Although Chord uniformly distributes values among peers (using consistent hashing), it does not succeed in fairly distributing access load in case of skewed query distributions.

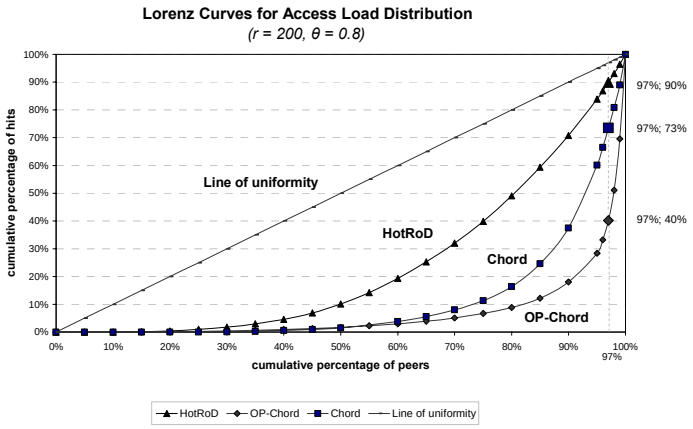

Fig 3. In Chord, OP-Chord, HotRoD, the top 3% heaviest peers receive almost 27%, 60%, 10% of total hits

We should mention that, in our experiments, the *upper access count threshold,* ρ_{max}, was set equal to the *average (value) access load* expected to be received by each peer in a uniform access load distribution. The latter is equal to $2 \cdot r$, as we briefly prove below.

Proof sketch. We assume that Q=20,000 queries request r values each, on average, and each peer is responsible for an average of $|DA|/N$=10 values. Therefore, we have $Q \cdot r \cdot N/|DA|$ hits uniformly distributed among N peers, and, thus an average of $Q \cdot r/|DA|$ hits per peer, which is equal to $2 \cdot r$, since $|DA|$ = 10,000. □

In our experiments, ρ_{max} was kept low (i.e. less than 50), which introduces a total of about 100% additional replicas. This is definitely realistic, given typical sharing network applications ([21], [24]); however, we stress that good load balancing can be achieved using even fewer replicas– see below.

4.4 Overhead Costs – Tuning Replication

An important issue is the degree of replication required to achieve a good load balancing performance. Therefore, we study the HotRoD architecture when tuning the parameter $\rho_{max,}$ the maximum allowed number of rings in HotRoD.

We ran experiments with different range spans, *r's*, and different access skew parameters, θ's. All experiments show that, as ρ_{max} increases, the numbers of peers' and tuples' replicas are increased till they reach an upper bound each (i.e. for r=50, θ=0.8, the upper bounds are 1449 for peers and 8102 for tuples).

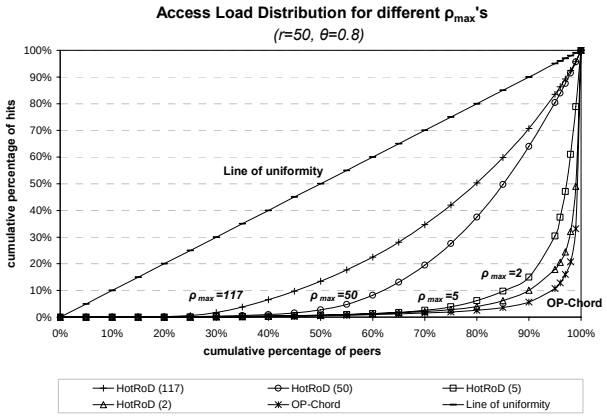

Fig 4. As ρmax increases, the Lorenz curves that illustrate the access load distribution come closer to the line of uniformity, which means that load imbalances and, thus, *G* are decreased

Figure 5 illustrates how different ρ_{max}'s affect the number of peers' and tuples' replicas for r=50 and θ=0.8. Specifically: for ρ_{max}=2, 11% of peers and 12% of tuples have been replicated; for ρ_{max}=5, 25% and 30% respectively; for ρ_{max}=10, 38% and 47%; for ρ_{max}=50, 85% and 103%. For high ρ_{max}'s, peers and replicas are heavily replicated, till ρ_{max} reaches 117 (as it was experimentally proven), beyond which there is no replication and, thus, there are no further benefits in load balancing.

Similar conclusions are drawn from experiments with different r's and θ's. In general, it holds that the lower the range span, r, or the higher the skew parameter, θ, the higher the upper limit of ρ_{max}. For example, for r=50 and θ=0.5, the upper limit of ρ_{max} is 90; for r=200, θ=0.8, it is 59; for r=200, θ=0.5, it is 23.

Figures 4 and 5 illustrate the role that ρ_{max} plays in the access load distribution. As ρ_{max} increases, the load imbalances are decreased, and G is decreased. Specifically, G is decreased as follows: for ρ_{max} = 2, G=0.92; for ρ_{max} = 5, G=0.88; for ρ_{max} = 50, G=0.64; for $\rho_{max} \geq 117$, G=0.50. On the other hand, the degree of replication (*RD*) for the number of tuple replicas is increased as follows: for ρ_{max} = 2, *RD* = 12%; for ρ_{max} = 5, *RD* =30%; for ρ_{max} = 50, *RD* =103%; for $\rho_{max} \geq 117$, *RD* =162%.

It is obvious that high values of ρ_{max} provide diminished returns in load balancing, although the degree of replication is very high. This means that we can achieve a very good access load balancing with low values of ρ_{max}, and thus, low overhead costs.

To recap: In terms of average hop-counts per range query, HotRoD ensures significant savings, which increase as the range span r, or the access skew parameter θ increases. At the same time, with respect to load balancing, HotRoD achieves its goal of involving the lightest hit peers to offer significant help to the heaviest hit peers, while the total replication overhead is no more than 100%.

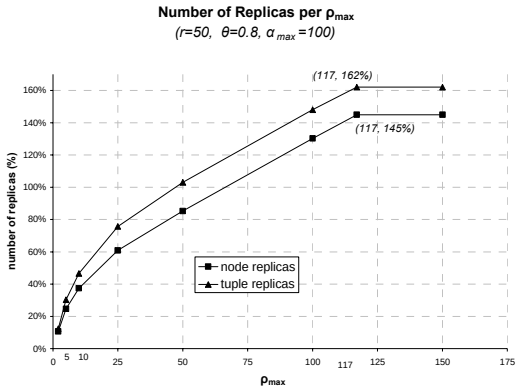

Fig 5. Tuning replication degree by $\rho_{\max}$

5 Related Work

There are quite a few solutions supporting range queries, either relying on an underlying DHT, or not. Some indicative examples of such DHTs solutions follow. Andrzejak and Xu ([2]) and Sahin, et al. ([19]) extended CAN ([17]) to allow for range query processing; however, performance is expected to be inferior compared to the other DHT-based solutions, since CAN lookups require $O(2 \cdot N^{1/2})$ hops, for a two-dimensional identifier space. Gupta et. al ([9]) propose an architecture based on Chord, and a hashing method based on a min-wise independent permutation hash function, but they provide only approximate answers to range queries. The system proposed in Ntarmos et al. ([14]) optimizes range queries by identifying and exploiting efficiently the powerful peers which have been found to exist in several environments. Ramabhadran et al ([16]) superimpose a trie (prefix hash tree – PHT) onto a DHT. Although their structure is generic and widely applicable, range queries are highly inefficient, since locality is not preserved. Triantafillou and Pitoura ([22]) outlined a Chord-based framework for complex query processing, supporting range queries. This was the substrate architecture of HotRoD, which we extended here to address replication-based load balancing with efficient range query processing. Although capable to support range queries, none of the above support load balancing.

Among the non-DHT solutions, the majority of them (such as Skip Graphs ([4]), SkipNet ([10]), etc) do not support both range queries and load balance. In a recent work ([3]), Aspnes et al provide a mechanism for providing load balancing using skip graphs. With the use of a global threshold to distinguish heavy from light nodes, they let the light nodes continue to receive elements whereas the heavy ones attempt to shed elements. However, many issues have been left unanswered, such as fault tolerance. Ganesan et al ([7]) propose storage load balance algorithms combined with distributed routing structures which can support range queries. Their solution may support load balance in skewed data distributions, but it does not ensure balance in skewed query distributions. BATON ([11]) is a balanced binary tree overlay network

which can support range queries, and query load balancing by data migration between two, not necessarily adjacent, nodes. In their Mercury system ([5]), Bharambe et al support multi-attribute range queries and explicit load balancing, using random sampling; nodes are grouped into routing hubs, each of which is responsible for various attributes.

In all the above approaches, load balancing is based on transferring load from peer to peer. We expect that this will prove inadequate in highly-skewed access distributions where some values may be so popular that single-handedly make the peer that stores them heavy. Simply transferring such hot values from peer to peer only transfers the problem. Related research in web proxies has testified to the need of replication ([23]). Replication can also offer a number of important advantages, such as fault tolerance and high availability ([13]) albeit at the storage and update costs. Besides, we have experimentally shown that storage and update overheads can be kept low, since we can achieve our major goals with a relatively small number of replicas.

Finally, an approach using replication-based load balancing, as ours, is [8], where a replication-based load balancing algorithm over Chord is provided; however, it appears that knowledge about the existence of replicas is slowly propagated, reducing the impact of replication. Besides, it only deals with exact-match queries, avoiding the most difficult problem of balancing data access loads in the presence of range queries.

6 Conclusions

This paper presents an attempt at concurrently attacking two key problems in structured P2P data networks: (a) efficient range query processing, and (b) data-access load balancing. The key observation is that replication-based load balancing techniques tend to obstruct techniques for efficiently processing range queries. Thus, solving these problems concurrently is an important goal and a formidable task. Some researchers claim that existing DHTs are ill-suited to range queries since their property of uniform distribution is based on randomized hashing, which does not comply with range partitioning (i.e. [5]). However, HotRoD succeeded in combining the good properties of DHTs (simplicity, robustness, efficiency, and storage load balancing) with range partitioning using a novel hash function which is both locality-preserving and randomized (in the sense that queries are processed in randomly selected – replicated - partitions of the identifier space).

We have taken an encouraging step towards solving the two key aforementioned problems through the HotRoD architecture. HotRoD reconciles and trades-off hop-count efficiency gains for improved data-access load distribution among the peers. Compared to base architectures our detailed experimentation clearly shows that HotRoD achieves very good hop-count efficiency coupled with a significant improvement in the overall access load distribution among peers, with small replication overheads. Besides, in parallel with the evaluation of HotRoD, we have introduced novel load balancing metrics (i.e. the Lorenz curves and the Gini coefficient) into the area of distributed and p2p computing, a descriptive and effective way to measure and evaluate fairness of any load distribution. Finally, HotRoD can be superimposed over any underlying DHT infrastructure, ensuring wide applicability/impact.

7 References

1. Aberer, K.: P-Grid: A self-organizing access structure for P2P information systems. In Proc of CoopIS (2001)
2. Andrzejak, A., and Xu, Z.: Scalable, efficient range queries for Grid information services. In Proc. of P2P (2002)
3. Aspnes, J., Kirsch, J., Krishnamurthy, A.: Load balancing and locality in range-queriable data structures. In Proc. of PODC (2004)
4. Aspnes, J., Shah, G: Skip graphs. In ACM-SIAM Symposium on Discrete Algorithms (2003)
5. Bharambe, A., Agrawal, M., Seshan, S.: Mercury: Supporting scalable multi-attribute range queries. In Proc. of SIGCOMM04 (2004)
6. Damgaard, C., and Weiner, J.: Describing inequality in plant size or fecundity. Ecology 81 (2000) pp. 1139-1142
7. Ganesan, P., Bawa, M., and Garcia-Molina, H.: Online balancing of range-partitioned data with applications to peer-to-peer systems. In Proc. of VLDB (2004)
8. Gopalakrishnan, V., Silaghi, B., Bhattacharjee, B., and Keleher, P.: Adaptive replication in peer-to-peer systems. In Proc. of ICDCS (2004)
9. Gupta, A., Agrawal, D., and Abbadi, A.E.: Approximate range selection queries in peer-to-peer systems. In Proc. of CIDR (2003)
10. Harvey, N., et al.: SkipNet: A scalable overlay network with practical locality preserving properties. In Proc.of 4th USENIX Symp. on Internet Technologies and Systems (2003)
11. Jagadish, H.V., Ooi, B.C., Vu, Q. H..: BATON: A balanced tree structure for peer-to-peer networks. In Proc. of VLDB (2005)
12. Karger, D., et al.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In Proc. ACM STOC (1997)
13. Mondal, A., Goda, K., Kitsuregawa, M.: Effective Load-Balancing via Migration and Replication in Spatial Grids. In Proc of DEXA 2003 (2003)
14. Ntarmos, N., Pitoura, T., and Triantafillou, P.: Range query optimization leveraging peer heterogeneity in DHT data networks. In Proc. of DBISP2P (2005)
15. Pitoura, T., Ntarmos, N., and Triantafillou, P.: HotRoD: Load Balancing and Efficient Range Query Processing in Peer-to-Peer Data Networks. Technical Report No. T.R.2004/12/05, RACTI (2004)
16. Ramabhadran, S., Ratnasamy, S., Hellerstein, J., Shenker, S.: Brief Announcement: Prefix Hash Tree. In Proc. of PODC (2004)
17. Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S.: A scalable content-addressable network. In Proc. ACM SIGCOMM (2001)
18. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In Proc. of Middleware (2001)
19. Sahin, O.D., Gupta, A., Agrawal, D., and Abbadi, A.E.: A peer-to-peer framework for caching range queries. In Proc. of ICDE (2004)
20. Saroiu, S., Gummadi, P., and Gribble, S.: A measurement study of peer-to-peer file sharing systems. In Proc. of MMCN (2002)
21. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., and Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In Proc. of SIGCOMM (2001)
22. Triantafillou, P., and Pitoura, T.: Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In Proc. of DBISP2P (2003)
23. Wu, K., and Yu, P.S.: Replication for load balancing and hot-spot relief on proxy web caches with hash routing. Distributed and Parallel Databases, 13(2) (2003) pp.203-220.
24. Zhao, Y.B., Kubiatowitcz, J., Joseph, A.: Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141 (2001)