Query Processing Over Peer-to-Peer Data Sharing Systems *

O. D. Şahin A. Gupta D. Agrawal A. El Abbadi
Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
{odsahin, abhishek, agrawal, amr}@cs.ucsb.edu

Abstract

Peer-to-peer systems are mainly used for object sharing currently, but they can provide the infrastructure for many other applications. In this paper, we extend the idea of object sharing to data sharing on a peer-to-peer system. We propose a method, which is based on the CAN [16] system, for efficiently evaluating range queries on such a system. The answers of the range queries are cached at the peers and are used to answer future range queries. The scalability and efficiency of our design is shown through simulation.

Keywords: Peer-to-peer Systems, Distributed Systems, Range Queries, Distributed Hashing, Distributed Caching.

1 Introduction

Peer-to-peer systems have been increasing in popularity in recent years as they are used by millions of users to share massive amounts of data over the Internet. These systems are generally used for file sharing, such as Napster [15], Gnutella [4] and KaZaA [10], which allow users to share their files with other users. There are two challenges to be resolved for sharing objects on a peer-to-peer system:

- **Data Location:** Given the name of an object, find the corresponding object's location.
- **Routing:** Once the possible location of the object is found, how to route the query to that location.

*This research was funded in parts by NSF grants EIA-0080134 and IIS-0223022.

Napster uses a centralized design to resolve these issues. A central server maintains the index for all objects in the system. New peers joining the system register themselves with the server. Every peer in the system knows the identity of the central server while the server keeps information about all the nodes and objects in the system. Whenever a peer wants to locate an object, it sends the request (name of the object) to the central server which returns the IP addresses of the peers storing this object. The requesting peer then uses IP routing to pass the request to one of the returned peers and downloads the object directly from that peer. There are several shortcomings of the centralized design of Napster. First of all, it is not scalable since the central server needs to store information about all the peers and objects in the system. Second, it is not fault tolerant because the central server is the single point of failure.

A different approach is followed by Gnutella to get around the problem of centralized design. There is no centralized server in the system. Each peer in the Gnutella network knows only about its neighbors. A flooding model is used for both locating an object and routing the request through the peer network. Peers flood their requests to their neighbors and these requests are flooded for a certain threshold. The problems associated with this design are the high overhead on the network as a result of flooding and the possibility of missing some requests even if the requested objects are in the system.

These designs, including Napster, Gnutella, and some other variants are referred to as *unstructured* peer-to-peer systems [9, 13], because the data placement and network construction are done arbitrarily in these systems. Another group of peer-to-peer designs are referred to as *structured* peer-to-peer systems and include systems such as CAN [16], and Chord [18]. These systems are based on implementing a distributed data structure called *Distributed Hash Table* (DHT) [16, 17, 18, 20] which supports a hash-table like interface for storing and retrieving objects.

CAN [16] uses a d-dimensional virtual address space for data location and routing. Each peer in the system owns a zone of the virtual space and stores the objects that are mapped into its zone. Each peer stores routing information about *O(d)* other peers, which is independent of the number of peers, N, in the system. Each object is mapped to a point in d-dimensional space and then the request is routed toward the mapped point in the virtual space. Each peer on the path passes the request to one of its neighbors which is closer to the destination in the virtual space. The average routing path has $O(dN^{1/d})$ hops which is the lookup time for exact match queries. Chord [18] assigns unique identifiers to both objects and peers in the system. Given the key of an object, it uses these identifiers to determine the peer responsible for storing that object. Each peer keeps routing information about $O(logN)$ other peers, and resolves all lookups via $O(logN)$ messages, where N is the number of peers in the system.

Since peer-to-peer systems have emerged as a powerful paradigm for data sharing over the Internet, a natural question arises if the power of peer-to-peer systems can be harnessed to support database functionality over peer-to-peer systems. Indeed, several research initiatives are underway to answer this question. For example, Gribble et al. [5] in their position paper titled

"What can peer-to-peer do for databases, and vice versa?" outline some of the complexities that need to be addressed before peer-to-peer systems can be exploited for database query processing. Similarly, in a recent paper Harren et al. [9] explore the issue of supporting complex queries in DHT-based peer-to-peer systems. Harren et al. report the implementation of database operations over CAN by performing a hash join of two relations using DHT. The underlying technique basically exploits the exact-name lookup functionality of peer-to-peer systems. [2, 7, 8, 11] discuss the issues of data integration and exchange between heterogeneous data sources in a peer-to-peer data management system.

There are potential applications that demand more complex query functionality than the basic "lookup by name" operation that the current P2P systems deliver. For example, the state governments may maintain demographic information for the population of the states. The servers maintain information like population distribution in the state by age, education, average family income, and health plan coverage etc. This information can be used by local county governments in the state for the purpose of planning development, educational and health projects. The planners at the local counties can ask queries like what is the percentage of people in the county that fall in the annual income range of $\$15,000 - \$20,000$. A key point to note is that unlike queries in traditional DBMS where the exact answers are required, the exactness of answer is not critical in these applications. A best-effort, statistically significant approximate answer will suffice. This example illustrates that a peer-to-peer data management system does not need to replicate every functionality of commercial DBMSs. The goal is to facilitate database-like query functionality over distributed data instead of just providing the exact-match lookup.

The work reported in this paper has similar goals as that of Harren et al. [9], in that we are interested in supporting database query processing over peer-to-peer systems. Most data-sharing approaches designed for peer-to-peer systems are concerned with exact lookup of data associated with a particular keyword. Our contention is that in order to achieve the larger goal of data-sharing in the context of a DBMS over peer-to-peer systems, we need to extend the current peer-to-peer designs that only support exact name lookups to range searches. Range searches or range selection is one of the fundamental functionality that is needed to support general purpose database query processing. The main motivation for this is that the `selection` operation is typically involved at the leaves of a database query plan and hence is a fundamental operation to retrieve data from the database. Assuming that such data partitions of a relation are extensively replicated at the peers due to prior queries, we would like to retrieve the data from the peer-to-peer system instead of fetching it from the base relation at the data source. To the best of our knowledge there is no work on distributed caching that caches ranges of data objects.

In this paper we propose to extend peer-to-peer systems to support more general queries on potentially more complex and more structured datasets. Unlike previous approaches for the design of distributed databases, our approach aims to support the management of loosely synchronized datasets that support more general queries in a peer-to-peer environment. Unfortunately, DHTs were designed for exact-match queries. In this paper, we will extend the idea of object sharing to data sharing and propose a method for efficiently answering range queries on a peer-to-peer data sharing system. Our general long term goal is to support the various types of complex queries used by DBMSs so that general peer-to-peer data support can be a reality.

The rest of the paper is organized as follows: Section 2 presents the formulation of the problem. Section 3 introduces the basic concepts of our design, which is explained in detail in Section 4. The experimental results are presented in Section 5. The last section concludes the paper and discusses future work.

2 Problem Formulation

Current peer-to-peer systems focus on object sharing and use object names for lookup. Our goal, on the other hand, is to design a general purpose peer-to-peer data sharing system. We consider a database with multiple relations whose schema is globally known to all peers in the system [1]. The peers cooperate with each other to facilitate the retrieval and storage of datasets. A straightforward extension and application of object naming is to use the relation name to locate the data in the system. However, such an approach will result in large amounts of data being stored redundantly and often unnecessarily throughout the network. A more desirable approach is to use peers to store the answers of the prior queries. Whenever a new query is issued, the peers are searched to determine if the query can be answered from the prior cached answers. This is similar to the known database problem often referred to as *Answering Queries using Views* [12]. Since the problem of answering queries using views is computationally hard even in centralized systems, we will instead focus at a restricted version by extending the exact lookup functionality of peer-to-peer systems to the range lookup of a given dataset. Hence, our goal is to develop techniques that will enable efficient evaluation of range queries over range partitions that are distributed (and perhaps replicated) over the peers in a peer-to-peer system.

The problem of performing range queries in a peer-to-peer system has also been investigated by Andrzejak and Xu [1]. Their solution uses hilbert curve mapping to partition data among peers in a way that contiguously distributes the data at peers. Due to partitioning there is no replication of data and hence failure of a peer can cause loss of data. Also, as the number of peers grows, the ranges stored at peers become smaller and smaller, and therefore, mutiple peers need to be contacted to answer a query range.

We assume that initially the database is located at a known site. All queries can be directed to this database. However, such a centralized approach is prone to overloading. Furthermore, the location of the data source may be quite remote in the peer-to-peer network, and hence response time may be slow. Our goal is for the peers to cooperatively store range partitions of the database, which are later used to respond to user queries. This will help in reducing the load on

[1]Research efforts such as Piazza [7, 8] and Hyperion [2, 11] are addressing the orthogonal problem of schema mediation in peer-to-peer data sharing systems.

the data source and hence also improve the response times to queries. Of course the challenge is how to track down where the various data range partitions are located. A straightforward approach would be to maintain a centralized index structure such as an interval tree that has the global knowledge about the locations of range partitions distributed over the network. However, such an approach would violate the key requirement of peer-to-peer systems, which is to ensure that the implementation is scalable, decentralized, and fault-tolerant.

Typically when an SQL query is formulated, a query plan is designed in the form of a query tree. A common optimization technique is to push the `selection` operations down to the leaves of the tree to minimize the data that has to be retrieved from the DBMS. A similar approach is used here to minimize the amount of data retrieved from other peers for range queries. Rather than retrieving all possible tuples from the actual database for each range query, the answers stored at the peers are searched to find a smaller set of tuples that is a superset of the query.

For example, if the answer of a range query $\langle 20, 35 \rangle$ for a given attribute is stored at a peer, then future queries such as $\langle 25, 30 \rangle$ can be answered using the result of $\langle 20, 35 \rangle$. Since the range $\langle 20, 35 \rangle$ subsumes the range $\langle 25, 30 \rangle$, it is enough to examine the tuples in the result of $\langle 20, 35 \rangle$, without any data retrieval from the database. In this way, less tuples are checked to compute the answer and all the tuples to be examined are retrieved directly from a single peer. This also decreases the load on the database since it is not accessed for every query. The problem can now be stated as follows:

Problem. Given a relation R, and a range attribute A, we assume that the results of prior range-selection queries of the form $R.A(LOW, HIGH)$ are stored at the peers. When a query is issued at a peer which requires the retrieval of tuples from R in the range $R.A(low, high)$, we want to locate any peers in the system which already store tuples that can be accessed to compute the answer.

In order to adhere to the peer-to-peer design methodology, the proposed solution for range lookup should also be based on distributed hashing. A nice property of DHT-based approach is that the only knowledge that peers need is the function that is used for *hashing*. Once this function is known to a peer, given a lookup request the peer needs to compute the hash value locally and uses it to route the request to a peer that is likely to contain the answer. Given this design goal, a naive approach would be to use a linear hash function over the range query schema, i.e., a linear hash function over low, $high$, or both low and $high$. A simple analysis reveals that such a hash function will only enable exact matches of given range requests. However we are also interested in the results of the range queries that may contain the given range, i.e., the ranges that are a superset for the given query range lookup. In the following sections we develop a DHT approach that enables range lookups that are not exact matches. In [6], we use locality preserving hash functions for range lookups that are based on similarity and hence provide approximate answers to range queries. In this paper, however, our technique ensures that if a range lookup yields an answer, it is a superset of the query range.

3 System Model

Our system is based on CAN [16], which was designed to store and retrieve individual data objects and supported exact match queries. In contrast, we need to support storage and retrieval of ranges of data objects, and therefore, we use a hash mapping specific to data ranges. The distribution of the stored ranges over the peers in the system may not be uniform, as it is query driven. Therefore, we have used a modified zone splitting strategy to obtain a better load distribution. Since we are not just looking for exact-matches but we may want super-ranges for a given query range, we have also introduced query forwarding strategies. In the following, we describe our hash mapping and other strategies in detail.

Our system uses a 2d virtual space in a manner similar to CAN. Given the domain $[a, b]$ of a one dimensional attribute, the corresponding virtual hash space is a two dimensional square bounded by the coordinates (a, a), (b, a), (b, b), and (a, b) in the Cartesian coordinate space. Figure 1 shows the corresponding virtual hash space for a range attribute whose domain is [20,80]. The corners of the virtual space are $(20, 20)$, $(80, 20)$, $(80, 80)$, and $(20, 80)$.

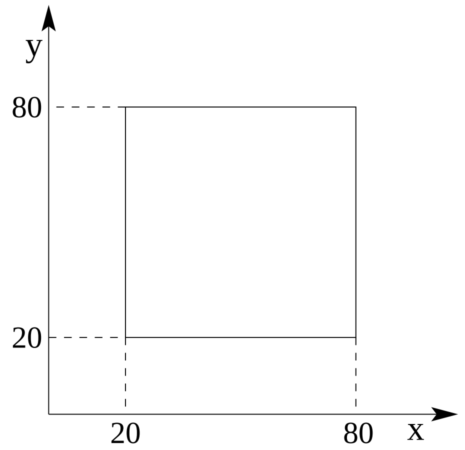


Figure 1: Virtual Range Lookup Space for a range attribute with domain $[20, 80]$

The virtual hash space is further partitioned into rectangular areas, each of which is called a *zone*. The whole virtual space is entirely covered by these zones and no two zones overlap. A zone can be identified by a pair $\langle(x_1, y_1), (x_2, y_2)\rangle$ where (x_1, y_1) is the bottom left corner coordinates whereas (x_2, y_2) is the top right corner coordinates. Figure 2 shows a possible partitioning of the virtual space shown in Figure 1. The virtual space is partitioned into 7 zones : *zone-1* $\langle(20,61),(30,80)\rangle$, *zone-2* $\langle(20,35),(80,50)\rangle$, *zone-3* $\langle(42,69),(80,80)\rangle$, zone-4 $\langle(20,50),(42,61)\rangle$, zone-5 $\langle(20,20),(80,35)\rangle$, zone-6 $\langle(42,50),(80,69)\rangle$, and *zone-7* $\langle(30,61),(42,80)\rangle$.

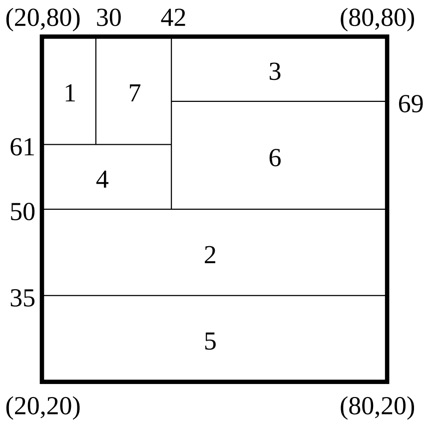


Figure 2: Partitioning of the virtual space shown in Figure 1

Each zone is assigned to a peer in the system. Unlike the original CAN, not all the peer nodes in the system participate in the partitioning. The nodes that participate are called the *active* nodes. Each *active* node *owns* a zone.The data source is responsible for the top-left zone, as the top-left corner corresponds to the complete database. The rest of the peer nodes, which do not participate in the partitioning, are called the *passive* nodes. Each passive node registers with one of the active nodes. All active nodes keep a list of passive nodes registered with them.

For the purpose of routing requests in the system, each *active* node keeps a *routing table* with the IP addresses and zone coordinates of its neighbors, which are the owners of adjacent zones in the virtual hash space. For example in Figure 2, the routing table of the owner of *zone-4* contains information about its four neighbors: *zone-1, zone-7, zone-6* and *zone-2*.

Given a range query with range $\langle q_s, q_e\rangle$, it is hashed to point (q_s, q_e) in the virtual hash space. This point is referred to as the *target point* of the query range. The target point is used to determine where to store the information about the answer of a range query as well as where to initiate range lookups when searching for the result of a range query. The zone in which the target point lies and the node that owns this zone are called the *target zone* and the *target node*, respectively. Therefore, the information about the answer of each range query is stored at the target node of this range.

Once a peer node gets the answer for its range query, it may choose to cache the answer and may inform the target node about it. The target node can store a pointer to the querying node or it can cache the result depending upon its cache space. In both cases, we say that the target node stores the result of this query. For example, according to Figure 2, the range query $\langle 50, 60 \rangle$ is hashed into *zone-6*, so the set of tuples that form the answer to this query may be stored at the node that owns *zone-6* or the node will store a pointer to the peer that caches the tuples in that range. Peers can choose one of the several well know caching policies, for example LRU, to manage their local cache space. The discussion of caching policies to manage the local storage at the peers is out of scope of this paper.

Since the start point and end point of a range is hashed to x and y coordinates respectively, the y coordinate of the target point is always greater than or equal to the x coordinate. Hence , the target point never lies below the $y = x$ line. Given two ranges $r_1 : \langle a_1, b_1 \rangle$, and $r_2 : \langle a_2, b_2 \rangle$ that are hashed to target points t_1 and t_2 in the virtual hash space, the following observations can be made:

1. If $a_1 < a_2$, then the x coordinate of t_1 is smaller than the x coordinate of t_2 and hence t_1 lies to the left of t_2 in the virtual space.

2. If $b_1 < b_2$, then the y coordinate of t_1 is smaller than the y coordinate of t_2 and hence t_1 lies below t_2 in the virtual space.

3. t_1 lies to the upper-left of t_2 if and only if range r_1 contains range r_2.

The third result can be concluded from the fact that moving along the negative x direction in the virtual hash space decreases the start point of the corresponding range while moving along the positive y direction increases the end point.

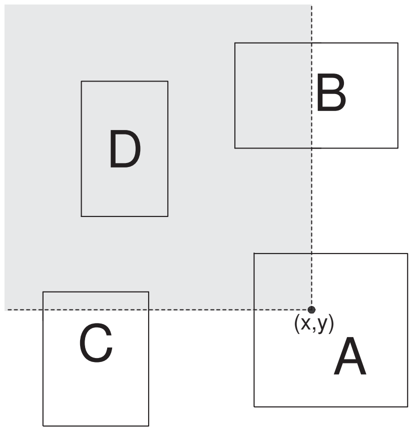


Figure 3: Range Hashing

Figure 3 shows a range query $\langle x, y \rangle$ that is hashed into zone A. Using the above observations, we can assert that if there is any prior range query result that contains $\langle x, y \rangle$, then it must have been hashed to a point in the shaded region. Any zone that intersects the shaded region is therefore a candidate for potentially containing a result for this query. In the figure, the zones A, B, C, and D intersect with the shaded region and may have a result that contains the given range $\langle x, y \rangle$.

The zone D in Figure 3 lies completely in the upper-left region of the target point (x, y). We call such zones *diagonal zones*, defined as follows:

Diagonal Zone. Consider a zone z bounded by coordinates $\langle (x_1, y_1), (x_2, y_2) \rangle$. We say that another zone z' bounded by $\langle (a_1, b_1), (a_2, b_2) \rangle$ is a diagonal zone of z if $a_2 \leq x_1$ and $b_1 \geq y_2$.

It is obvious that a zone cannot have a diagonal zone if it lies on the left or top boundary of the virtual space. It is also possible that a zone may have no diagonal zone even if it has many zones to the upper left of itself. Figure 4 shows such a case where the *zone-7* at the bottom right corner has no diagonal zones.

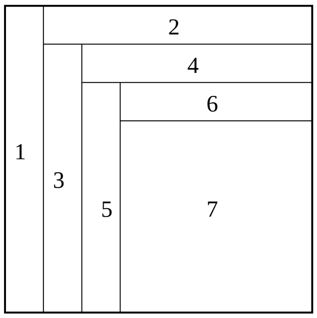


Figure 4: No Diagonal Zones

Diagonal zones are of particular interest since they have a high probability of containing a range which is a superset of the desired answer set. This is the case because every range that maps to a point in the diagonal zone contains the query range, and so if the diagonal zone has a cached result, then it is a superset of the desired answer. As the number of zones in the system increases, the possibility of finding a diagonal zone for a zone also increases.

4 Distributed Range Hashing

In this section we describe the basic components that support the distributed implementation of range hashing. We assume that there is a set of computing nodes which participate in the distributed implementation of the range hash table (RHT). For simplicity, we are assuming that the range hash table is based on a relation R for a specific range attribute A. In section 4.4 we explain how the system can be generalized to handle multiple attribute range queries. If queries on various relations need to be supported, we assume a separate instance of an appropriate RHT will be maintained for each relation.

The nodes participating in the system are in one of the two modes: *active/passive*. Initially, only one active node (the data source) manages the entire virtual hash space. Other nodes become active as the work load on the active nodes increases. Next, we describe how zones in the virtual hash space are maintained on peers. Finally, we present the details of range query lookup processing in the system.

4.1 Zone Maintenance

The partitioning of the virtual hash space into zones is at the core of both the data location and routing algorithms. Initially the entire hash space is a single zone and is assigned to the data source which is the only active node. The partitioning of the hash space is dynamic and changes over time as the existing zones split and new zones are assigned to passive nodes that become active and take responsibility for the new zones.

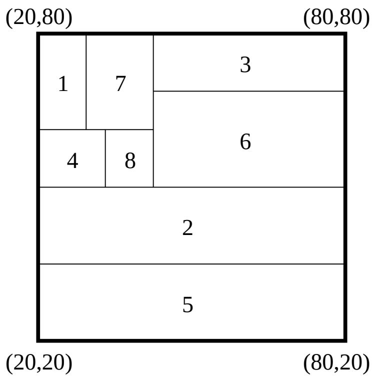


Figure 5: Partitioning of the virtual hash space after *zone-4* of Figure 2 splits

The decision to split is made by the owner of the zone. Whenever a zone needs to split, the owner node discovers a passive node either through its own passive node list or by forwarding the request for a passive node to its neighbors. The owner node then contacts one of the passive nodes and assigns it a portion of its zone by transferring the corresponding results and neighbor lists. A zone may split because of the following two reasons. First, it may have to answer too many queries. In this case, it splits along a line which results in an even distribution of stored answers as well as an even spatial distribution of the zone. Second, it may be overloaded because of routing queries, as larger zones are more likely to fall in the path of a query route. Therefore, the zone splits into equal halves along the longer side to reduce the routing load. The outline of the split operation is shown in Algorithm 1. The new peer is assigned the right partition if the zone splits parallel to *y-axis*, or the bottom partition if it splits parallel to *x-axis*. Figure 5 shows the partitioned zones after *zone-4* in Figure 2 splits parallel to the *y-axis* and the new peer is assigned *zone-8*.

Algorithm 1 Split a zone

```
if the zone needs to be divided because of answering load then
    Find x-median and y-median of the stored results.
    Determine if a split at x-median (parallel to y-axis) or a split at y-median
    (parallel to x-axis) results in even distribution of stored answers and the
    space.
else
    The split line is the midpoint of the longer side.
end if
Compute new coordinates of this zone and the new zone according to the split
line.
Assign the new zone to a passive node.
for all result points stored at this zone do
    if the result point is mapped to the new zone then
        Remove from this node and send to the new node.
    end if
end for
Transfer data tuples falling into the new zone to the new node.
for all neighbors of this zone do
    if it is a neighbor of the new zone then
        Add it to the neighbor list of new node.
        Inform the neighbor of new zone.
    end if
    if it is no longer a neighbor of this node then
        Inform neighbor to update its list.
        Remove from the neighbor list of this node.
    end if
end for
Add new node to the neighbor list of this node.
Add this node to the neighbor list of new node.
```

4.2 Query Routing

When searching for the answer of a range query, the first place to look for cached results is the target zone of this range. Therefore whenever a range query is issued, it is routed toward its target zone through the virtual space. Starting from the requesting zone, each zone passes the query to an adjacent zone until it reaches its target zone. Using its neighbor lists and the target point of the query, each node on the route passes the query to one of its neighbors whose coordinates are the closest to the target point in the virtual space. Algorithm 2 presents an outline of the routing procedure.

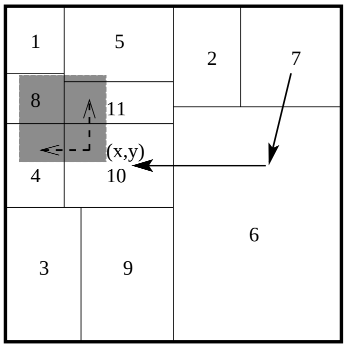


Figure 6: Routing and forwarding in virtual space. The shaded region shows the *Acceptable Region* for the query

Figure 6 shows how a query is routed in the system. The range query is initiated at *zone-7* and then routed through *zone-6* to its target zone, *zone-10*. The range queries in the system can be initiated from any zone. Since passive nodes do not participate in the partitioning, they send their queries to any of the active nodes from where the queries are routed toward the target zone.

An estimation of the average routing distance for processing range queries in the proposed model is presented in Appendix Appendix A. The analysis shows that the average routing path length in an equally partitioned hash space is $O(\sqrt{n})$, where n is the number of zones in the system.

Algorithm 2 Routing

if the query range maps to this zone **then**
  Return this zone.
else
  for all neighbors of this zone **do**
    Compute the closest Euclidean distance from the target point of the query to the zone of this neighbor in the virtual space.
    if this is the minimum distance so far **then**
      Keep a reference to this neighbor.
    end if
  end for
  Send the query to the neighbor with minimum distance from the target point in virtual space.
end if

4.3 Forwarding

Once a query reaches the target zone, the stored results at this zone are checked to see if there are any results whose range contains the query range. If such a result is found locally then it is directly sent to the querying peer. If there is a pointer to a peer node that stores a superset range then the address of that peer is forwarded as the answer and the querying peer can contact this peer to obtain the answer. Even if there is no such local result, it is still possible that some other zones in the system do have such a result; so the search should be forwarded to other zones. Fortunately the search space can be pruned at this point. As a result of the mapping scheme, only the zones that lie to the upper left of the target point can have a result containing the given range. So if the result is not found locally at the target node, the query is forwarded to the left and top neighbors that may contain a potential result. Those nodes also check their local results and may forward the query to some of their neighbors in a recursive manner.

Figure 6 shows how a query can be forwarded in the system. If the range query cannot be answered at its target zone, *zone-10*, then it is forwarded to *zone-4* and *zone-11* which may have a result for the query. Note that forwarding is only used if the query cannot be answered at the target zone. We describe two strategies for forwarding queries.

4.3.1 Flood Forwarding

A naive approach to forwarding queries is flood forwarding. Flood forwarding is similar to flooding the query to the neighbors that lie in the upper and left side of the virtual space. We use a parameter called *acceptable fit* to control forwarding, which is a real value between 0 and 1. It is used to determine how big an answer range is acceptable for a given query and therefore also determines how far the forwarding will continue. It defines an allowed offset for query range such that $offset = AcceptableFit \times |domain|$, where $|domain|$ is the length of the domain of the range attribute. The acceptable cached results for the range query $\langle low,\ high \rangle$, are then those that both contain the query range and are subsumed by the range $\langle low - offset,\ high + offset \rangle$. The square defined by these offsets and the target point is referred to as *Acceptable Region* as shown by the shaded area in Figure 6. Each node that receives a forwarded query checks its local results to find a result whose range is within the allowed offset of the query range. If it finds such a result, it notifies the querying peer and stops forwarding. Otherwise it forwards the query to its neighbors which may have a result within the given offset of the query range. If *acceptable fit* is set to 0, then only the target zone of the range query is checked and the query is not forwarded to any neighbor. Note that setting *acceptable fit* to 0 means that only exact answers can be matched for a given query. If, on the other hand, it is set to 1, then the query can be potentially forwarded to all zones that are likely to have a result for the query; i.e., all the zones which have some point that lies on the upper left of the target point of the query.

4.3.2 Directed Forwarding

Flood forwarding may result in too much communication overhead on the network. Therefore, instead of flooding the query to all the neighbors in upper and left directions, *directed forwarding* picks up a neighbor in the upper left region of the target point in the following manner. Out of all the neighbors that fall in the left and upper region for the target point, the neighbor that has the highest overlap area with the acceptable region is forwarded the query request. A query can specify a limit d on the directed forwarding. Whenever a query is forwarded, the limit d is decremented. When d becomes 0, the querying peer is notified to directly contact the data source. Directed forwarding is used in conjunction with the *acceptable fit* parameter, as described in Algorithm 3. Directed forwarding is useful because a querying peer can potentially bound the response time by specifying a limit on the number of hops during forwarding. Similar approaches have been explored by Lv et al. [14] and Yang et al. [19]. Freenet [3] implements a directed DFS in an unstructered P2P system.

Algorithm 3 Directed Forwarding (S, f, d)

$acceptableFit \leftarrow f$
$directedLimit \leftarrow d$
Compute *acceptableRegion* using $acceptableFit$.
if there is an answer which is in *acceptableRegion* **then**
  Return this answer.
end if
if $directedLimit > 0$ **then**
  Pick a neighbor n that has the highest area overlap with the *acceptableRegion* of target point and has not been visited earlier.
  Add neighbor n to the set of visited zones S.
  $directedLimit \leftarrow directedLimit - 1$
  Forward the query to neighbor n.
else
  Return answer not found to query source.
end if

4.4 Discussion

An important routing improvement in the system is *Lookup During Routing.* Since the requesting zone and the target zone can be at any position with respect to each other (they can actually be the same zone), it is possible that a zone on the path from the requesting zone to the target zone may already have a result containing the query. The system can be improved so that every zone on the route checks its local results if it may have a possible result. If the result is found, then the query is not routed any further and the result is returned to the requesting node. This way, some routing and forwarding can be avoided. The routing path decisions can be changed so that the routed queries follow a path that may have zones with possible results. The effect of *Lookup During Routing* on the system is shown in section 5.3.

Another possible modification to the system is to allow the nodes to ask warm up queries when they participate in the partitioning. When a passive node is assigned a zone, it may compute and cache the result of the query whose range is mapped to the upper left corner of its zone in order to warm up its cache. This way, further queries mapped to this zone will always be answered without forwarding because the range of the warm up query contains all ranges mapped to this zone. Although this improvement is not always possible, for example when the node is busy or the result of such a query is very large, it greatly improves the performance.

Some of the improvements for CAN[16] are also applicable for our approach. Multiple realities, better routing metrics, overloaded zone, and topology-sensitive partitioning can be incorporated into the system. Node departures can also be handled in the same way. For soft departure, the active node hands over its zone and other necessary data to a passive node. If, however, an active node fails, one of its neighbors takes over its zone and assigns it to a passive node. To be able to detect failed nodes, active nodes send periodic update messages to their neighbors.

Although our system is designed for answering range queries, it can also answer exact queries. Exact match queries can be answered by setting the start and end points of the range to the exact value and then querying the system with this range. For example, in order to search for the tuples with range attribute A=20, the system is queried for the range $\langle 20, 20 \rangle$.

Updates of tuples can be incorporated into the system in the following manner. When a tuple t with range attribute $A = k$ is updated, an update message is sent to the target zone of the range $\langle k, k \rangle$. Since tuple t is included in all the ranges $\langle a, b \rangle$ such that $a \leq k$ and $b \geq k$, the update message is forwarded to all zones that lie on the upper left of the target zone. Each zone receiving an update message, updates the corresponding tuple in the local results accordingly. Note that the receiving zones may be storing pointers to peer nodes with actual data instead of caching it. In that case, the zone forwards the update message to the actual node(s) storing the data tuple. All nodes that cache the tuple t will receive the update message and hence will update the tuple value in the stored data partition. The problem with this scheme is that the zones on the upper left part of the virtual space get more update messages than the others. We plan to explore methods that will mitigate this problem. For example, batching of multiple updates is one possible solution.

Our solution elegantly generalizes to multiple attribute range selection queries. Consider a relation with n attributes. The range selection queries over this relation are mapped into a $2n$-dimensional virtual space. Let us say that L_i and H_i represent the limits of the domain for attribute i. A range selection query over the n attributes can be written as $\langle l_1, h_1 \rangle, \langle l_2, h_2 \rangle, \ldots, \langle l_n, h_n \rangle$. If an attribute is missing in the range query, then we use the domain of the attribute as the selection range for that attribute. The query is mapped to the point $(-l_1, h_1, -l_2, h_2, \ldots, -l_n, h_n)$ in the $2n$-dimensional space. The first dimension of the virtual space is bounded by $(-H_1, -L_1)$ whereas the second dimension is bounded by (L_1, H_1). Note that these two dimensions correspond to the first attribute. Similarly the third and fourth dimensions of the virtual space are bounded by $(-H_2, -L_2)$ and (L_2, H_2) respectively, and so on. In this case, the DHT will be a $2n$-dimensional CAN. In 2d this mapping means that a sub-range R_1 of a range R_2 is always on the lower left of R_2. Therefore, a superset can be found by moving towards increasing coordinates for each axis. In the general case, if the result is not available at the target zone, the query needs to be forwarded only in the increasing direction for all the dimensions of the space. The routing algorithm for the $2n$-dimensional virtual space remains the same. In this paper, we have used a slightly different mapping for the 2d case, which maps both ends of a range to positive coordinates, in order to keep the discussion simple.

5 Experimental Results

We implemented a simulator in Java and then tested various aspects of our design. In this section, we present the test results. All experiments were performed on a machine with dual Intel Xeon 2GHz processors and 1GB of main memory, running Linux RedHat 8.0.

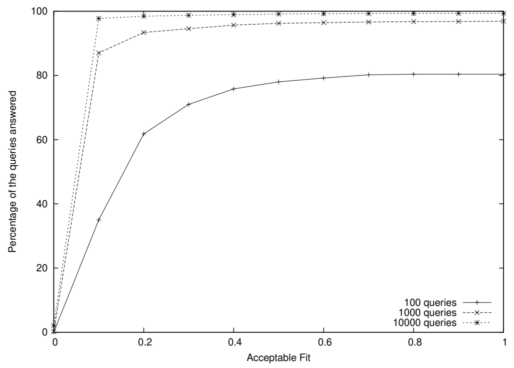
In the experiments, a zone splits when the number of stored results exceeds a threshold value, which is called the *split point*, or the routing load on a node goes higher than a specific threshold. The reported values are averaged over 5 runs. Unless mentioned otherwise, the following default values are used in all of the experiments:

- The system is initially empty. (There is only one zone in the system.)
- Split point is 5, i.e., at most 4 query results are maintained per peer.
- Routing threshold is 3 queries in a second.
- Range queries are distributed uniformly at random.
- Domain of the range attribute is $\langle 0, 500 \rangle$.

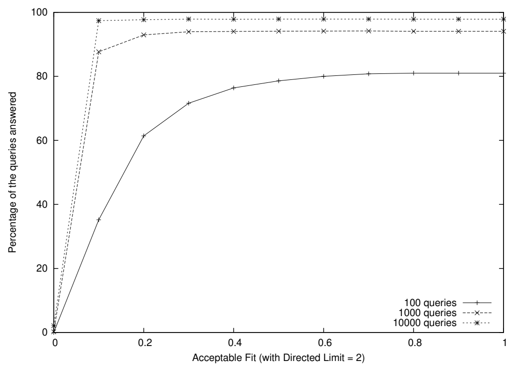
5.1 System Performance

The performance of the system can be determined in terms of the ratio of the range queries that are answered using prior answers stored in the system. In these experiments, we have measured the system performance using query sets of sizes 100, 1000 and 10,000. For each size we have used 5 different query sets generated randomly. The final number of active peers in these experiments is limited to 10% of the total number of queries. Once the maximum number of active peers is reached in the system, all active peers stop caching after they cache 4 range queries and the cached results are not expired.

Figure 7(a) shows the percentage of the answered range queries as a function of the acceptable fit when flood forwarding is used in the system. When the acceptable fit is 0, no forwarding is used and only the target zone is checked



(a) System performance with flood forwarding



(b) System performance with directed forwarding

Figure 7: System Performance

for each query. With 0 acceptable fit, only exact answers can be matched for a given range query, and therefore, the success ratio is almost 0. When forwarding is enabled (even if it is set to a small value such as 0.1), there is a great improvement in performance. With 100 queries, changing the acceptable fit from 0 to 0.1 improves the performance to 35%, whereas the performance goes up to 97.75% under the same conditions with 10,000 queries.

If the acceptable fit is set to 1, then every zone that may have a possible result for the query is searched and a stored result that contains the query range is found if there exists any. When forwarding limit is set to 1 for 10,000 queries, 99.36% of the queries are answered using cached results. Note that the success rate does not improve much after the acceptable fit is 0.5. In the next experiment, we introduced directed forwarding with a directed limit of 2 hops.

We measured the success rate of the system for various acceptable fit values for directed forwarding. The results have been summarized in Figure 7(b). Even with a small limit of 2 for directed forwarding the success rates are very similar to flood forwarding. We can make two important observations from Figures 7(a) and 7(b):

- *The probability of finding answers to range queries improves as the acceptable fit is increased.* This is quite clear since increasing the acceptable fit allows a larger set of cached ranges to match a given query range.
- *The probability of finding answers to range queries improves as the number of queries increases.* As the number of queries is increased, more results are stored in the system and the possibility of finding a result for a query gets higher.

We also measured the effect of directed limit on performance when the acceptable fit is 0.5. Figure 8 presents the results of the experiment. It can be seen that a directed limit of 4 is sufficient for most general purposes. Directed forwarding finds the answer within few hops of the target zone, which shows that our directed forwarding strategy is effective.

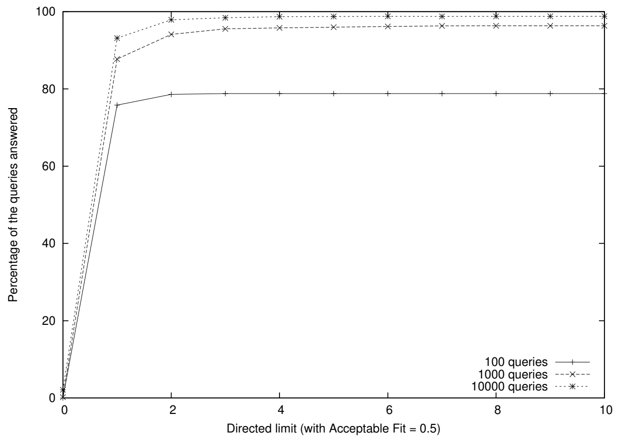
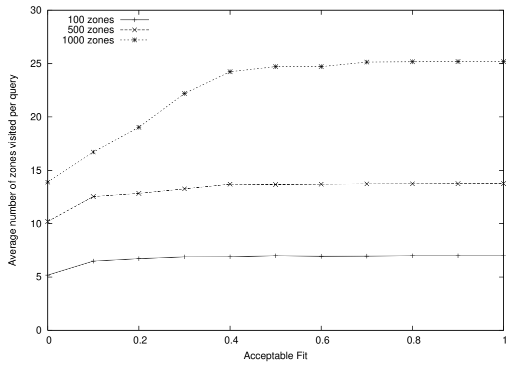


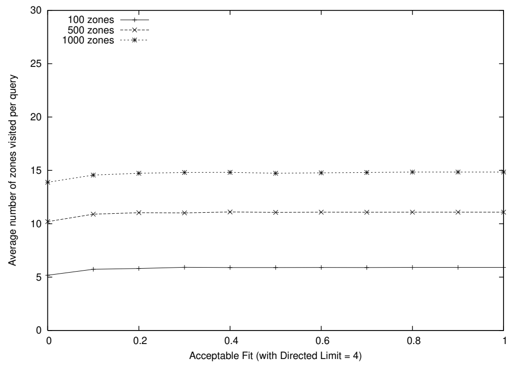
Figure 8: Effect of directed limit on system performance

5.2 Routing Performance

The routing performance is measured in terms of the average number of zones visited for answering a query. We have measured the effects of flood forwarding and directed forwarding on the average number of zones visited to answer queries. We simulated the system with the number of peers in the system set to 100, 500 and 1000. We started measuring the path lengths after the number of active peers in the system has reached the desired value. We ran 5 sets of 10,000



(a) Flood forwarding



(b) Directed forwarding

Figure 9: Routing Performance

uniformly randomly distributed range queries and averaged the route lengths over those runs. When the acceptable fit is 0, the result is the average number of zones visited during routing. If forwarding is enabled, it also includes the zones visited during forwarding.

Figures 9(a) and 9(b) show the average number of zones visited to answer queries when flood forwarding and directed forwarding are used respectively. Directed forwarding contacts less number of zones and it also scales well with increasing number of peers in the system. The average path length increases about 2.5 folds with an increase of 10 folds in the number of zones for directed forwarding. Even with flood forwarding it is less than square root of the number of peers in the system.

From the performance and path length experiments, we conclude that directed forwarding is a significant improvement over flood forwarding. It provides similar success in locating answers by contacting less number of zones.

5.3 Lookup During Routing

One of the improvements to the system is to implement *Lookup During Routing* so that the results for the queries may be found while they are being routed to their target zone. If a result for the query is found on its way to its target zone, it is immediately returned to the querying peer resulting in less number of visited zones. In the experiments, we have used 5 sets of 10,000 queries and the limit for directed forwarding is 2 hops. The final number of active peers in the system is limited to 1000.

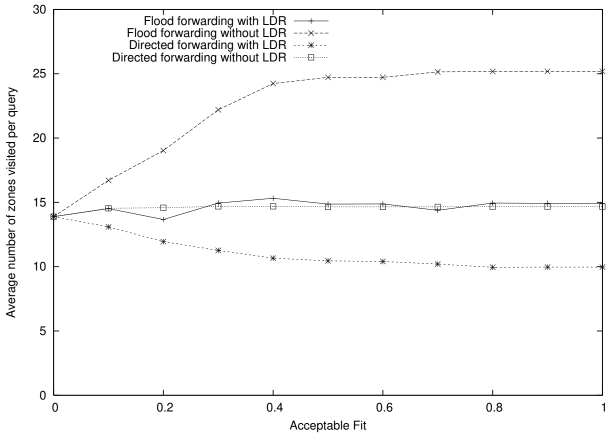


Figure 10: Effect of *Lookup During Routing* on the number of visited zones

Figure 10 shows the number of visited zones per query when lookup during routing is used. Lookup during routing substantially reduces the number of zones visited. In our experiments we observed that around 40% of the queries are answered during routing when acceptable fit is non-zero. Notice that the number of zones visited in the case of flood forwarding with lookup during routing wobbles with increase in acceptable fit. The reason for this behavior is that as acceptable fit is increased, the probability of finding an answer during routing increases. But if the answer is not found on the route, then the flood forwarding results in contacting more zones.

5.4 Selectivity

Figure 11 shows the performance of the system when query ranges are restricted to certain maximum lengths. The domain of the range attribute is changed to 0-10,000 in order to avoid the repetitions of queries when the selectivity is small.

In the figure, *Selectivity k%* means that the length of any queried range is less than or equal to $(k * |domain|/100)$ where $|domain|$ is the length of the domain and equals 10,000 in this case. For example, with 0.1% selectivity, all query ranges have length less than or equal to 10 since $0.1 * 10000/100 = 10$. 100% selectivity is the same as no selectivity since the query ranges can have any possible length. When creating the range queries, the start points of the ranges are selected uniformly from the domain of the range attribute and then the length of the range is determined randomly according to the selectivity. For each selectivity value, we used 3 sets of 10,000 queries.

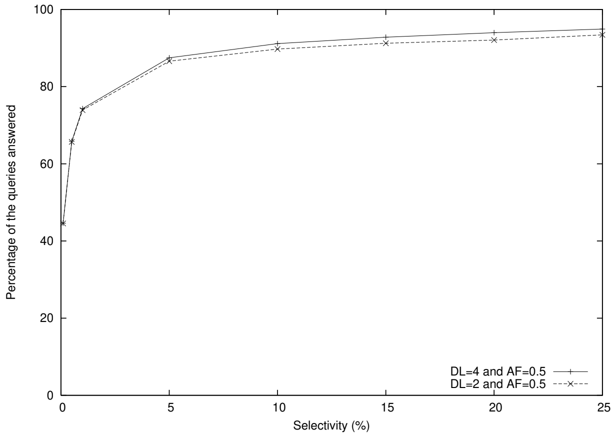


Figure 11: Effect of *selectivity* on system performance

In the graphs, AF stands for acceptable fit and DL stands for directed limit. As seen from Figure 11, the percentage of queries answered decreases as the selectivity gets smaller. That is because restricting the query ranges to a smaller length makes it harder to find prior results that contain a given range. When the selectivity is small, a query is looking for a very specific range. All the prior queries have also been quite specific. Hence the probability that the current query is contained in one of the prior queries is low, which explains the observed behavior. Low selectivity negatively impacts the query answering capability of the system.

5.5 Load Distribution

In the following experiments we measured the load on the peers in the system. The load on the peers can be classified in two categories. *Answering Load* measures the total number of queries that a peer has to answer. *Message Load* measures the total number of messages that a peer needs to process due to routing or forwarding.

We ran a set of 10,000 queries in a system of 1000 active peers. The number of zones remains constant during the run. Figure 12 shows the total load which is the summation of answering and message load on each peer. We have sorted the load values on all the peers after the completion of the run in increasing order. The marked points on the curves represent the average total load over all the peers for the corresponding run. The maximum load for each run is almost 3.5 times the average load. But the maximum load on any peer for all the runs is around 0.35% of the total load on the system. The load curve for the case with acceptable fit 0.5 and directed limit 4 is almost the same as the one with acceptable fit 1 and directed limit 4. Therefore, we have omitted the curve for 0.5 from the graph.

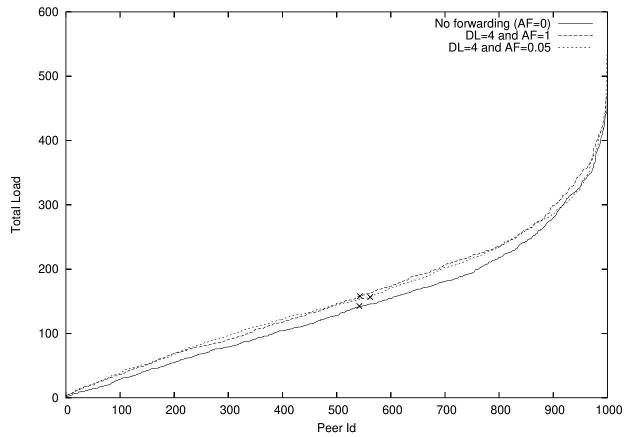


Figure 12: Load Distribution

In our system, the zones in the upper left region of the virtual space are responsible for long ranges. An initial impression is that the zone on the top-left would be overloaded. However, our experiments have shown that this is not the case. We noted during these experiments that the load on the top-left zone, which is maintained by the data source, is very close to the average load. We conclude that the system does not incur much overhead over the source and reduces its burden in answering queries which is evident from the performance experiments.

6 Conclusions and Future Work

Peer-to-peer systems are gaining in importance as they distribute information and connect users that are distributed across the globe. The true distributed systems of today need to facilitate this world wide retrieval and distribution of data. So far most peer-to-peer attempts have been restricted to exact match lookups and therefore are only suitable for file-based or object-based applications. This paper represents a first step toward the support of a more diverse and richer set of queries. Databases are a natural repository of data, and our enhanced CAN-based system supports the basic range (or selection-based) operation. Our approach is simple and very promising. We have shown how to exploit the CAN approach to support range queries and have demonstrated that it successfully scales using a variety of performance studies. Our system greatly reduces the burden of answering queries from the data source with only a little overhead.

Our future efforts are directed towards a design of a complete peer-to-peer database system. In the short term, we plan to explore multi-attribute range queries as well as non-integer based domains in detail.

References

[1] Artur Andrzejak and Zhichen Xu. Scalable, efficient range queries for grid information services. In *Proceedings of the 2nd IEEE P2P*, 2002.

[2] P. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data management for peer-to-peer computing: A vision. In *Proceedings of the 5th WebDB*, 2002.

[3] Freenet. http://freenet.sourceforge.net/.

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

[5] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can peer-to-peer do for databases, and vice versa? In *Proceedings of the 4th WebDB*, 2001.

[6] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of the 1st CIDR*, 2003.

[7] A. Halevy, O. Etzioni, A. Doan, Z. Ives, J. Madhavan, L. McDowell, and I. Tatarinov. Crossing the structure chasm. In *Proceedings of the 1st CIDR*, 2003.

[8] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *Proceedings of the 19th ICDE*, 2003.

[9] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of the first International Workshop on Peer-to-Peer Systems*, 2002.

[10] KaZaA. http://www.kazaa.com/.

[11] Anastasios Kementsietsidis, Marcelo Arenas, and Renee J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *Proceedings of the 2003 ACM SIGMOD.*

[12] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views (extended abstract). In *Proceedings of the 14th ACM PODS*, pages 95–104. ACM Press, 1995.

[13] Q. Lv, S. Ratnasamy, and S. Shenker. Can heterogeneity make gnutella scalable? In *Proceedings of the first International Workshop on Peer-to-Peer Systems*, 2002.

[14] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th ICS*, 2002.

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

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 ACM SIGCOMM.*

[17] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware 2001.*

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

[19] Beverly Yang and Hector Garcia-Molina. Improving search in peer-to-peer networks. In *Proceedings of the 22nd ICDCS*, 2002.

[20] 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.

Appendix A. Average Routing Path Length

We now estimate the average routing distance for processing range queries in the proposed model. Let us assume that the 2d virtual hash space is partitioned into n equal sized zones. If $n = m^2$, then each axis is divided into m equal sized parts. Every path from a zone to another zone constitutes of steps in x and y directions (see Figure 13, the path is shown as a dotted line).

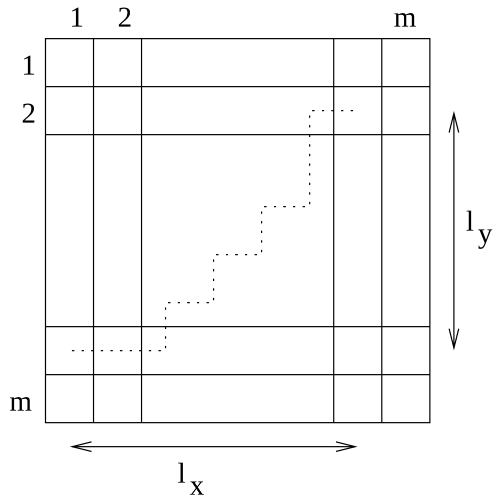


Figure 13: A Path in 2d virtual hash space divided into equal sized zones

In Figure 13, l_x is the number of steps in x direction and l_y is the number of steps in y direction. Then the total path length $l = l_x + l_y$. The shortest possible path length is 0 and the longest path length is $2m - 2$, which is $m - 1$ steps in either direction. Now, we will compute how many ways a path of length l can be generated. When $l < m$, $0 \leq l_x \leq l$ and $l_y = l - l_x$. A path of length l_x in the x direction can be generated in $m - l_x$ ways, and a path of length l_y in the y direction can be generated in $m - l + l_x$ ways. Hence the total number of ways in which a path of length l can be generated when $l < m$ is given by $S1$:

$$\begin{aligned} S1 &= \sum_{l_x=0}^{l} (m - l_x)(m - l + l_x) \\ &= -\frac{l}{6} + \frac{l^3}{6} - lm - l^2 m + m^2 + lm^2 \qquad (1) \end{aligned}$$

Therefore, the total number of paths whose length is less than m is:

$$E1 = \sum_{l=0}^{m-1} S1$$

$$= \frac{m}{12} + \frac{7m^2}{24} + \frac{5m^3}{12} + \frac{5m^4}{24} \quad (2)$$

And, the weighted sum of all path lengths less than m is:

$$\begin{aligned} E2 &= \sum_{l=0}^{m-1} S1 \cdot l \\ &= -\frac{m}{30} - \frac{m^2}{12} - \frac{m^3}{12} + \frac{m^4}{12} + \frac{7m^5}{60} \quad (3) \end{aligned}$$

When $m \leq l \leq 2m-2$, we have $l_x \leq m-1$ because that is the maximum distance that can be traveled in any direction. Since $l_y \leq m-1$, therefore, $l_x \geq l-m+1$. Hence the total number of ways in which a path of length l can be generated when $m \leq l \leq 2m-2$ is given by $S2$:

$$\begin{aligned} S2 &= \sum_{l_x=l-m+1}^{m-1} (m-l_x)(m-l+l_x) \\ &= \frac{l}{6} - \frac{l^3}{6} - \frac{m}{3} + l^2m - 2lm^2 + \frac{4m^3}{3} \quad (4) \end{aligned}$$

The corresponding total number of paths and weighted sum are given by the following expressions:

$$\begin{aligned} E3 &= \sum_{l=m}^{2m-2} S2 \\ &= -\frac{m}{12} - \frac{m^2}{24} + \frac{m^3}{12} + \frac{m^4}{24} \quad (5) \\ \textit{and, } E4 &= \sum_{l=m}^{2m-2} S2 \cdot l \\ &= \frac{m}{30} - \frac{m^2}{12} - \frac{m^3}{12} + \frac{m^4}{12} + \frac{m^5}{20} \quad (6) \end{aligned}$$

The average path length $\bar{l}$ is therefore given by the weighted sum over all possible path lengths divided by total number of paths:

$$\begin{aligned} \bar{l} &= \frac{\sum_{l=0}^{m-1} S1.l + \sum_{l=m}^{2m-2} S2.l}{\sum_{l=0}^{m-1} S1 + \sum_{l=m}^{2m-2} S2} \\ &= \frac{E2+E4}{E1+E3} \\ &= \frac{2}{3}(m-1) \quad (7) \end{aligned}$$

The average path length in an equally partitioned hash space is therefore $O(\sqrt{n})$.