Approximate Range Selection Queries in Peer-to-Peer Systems

Abhishek Gupta Divyakant Agrawal Amr El Abbadi

University of California
Santa Barbara, CA 93106
USA
{abhishek, agrawal, amr}@cs.ucsb.edu

Abstract

We present an architecture for a data sharing peer-to-peer system where the data is shared in the form of database relations. In general, peer-to-peer systems try to locate exact-match data objects to simple user queries. Since peer-to-peer users generally tend to submit broad queries in order to find data of their interest, we develop a P2P data sharing architecture for computing approximate answers for the complex queries by finding data ranges that are similar to the user query. Thus this paper represents the first step towards solving the general range lookup problem over P2P systems instead of exact lookup operations.

1 Introduction

In recent years there has been a significant interest in peer-to-peer data sharing systems. Most of the research has concentrated on sharing of file objects such as music or video files. In general, users who wish to participate in a peer-to-peer system register their machines to become part of the peer-to-peer system. Hence, a user machine becomes a peer node in the system. Users at a peer can submit a query string that is the name of the file they are looking for. The system finds a peer that has a copy of the requested object and directs the querying peer to the peer that can provide the requested object. There are two design challenges that arise in the context of building P2P systems. The application level problem is to locate a peer that stores the requested object. The system level problem is to route the query from the requesting peer to the peer where the object is stored. The critical issue is to locate a peer that can provide the requested object.

One solution to this problem is building a centralized index (e.g. Napster [12]). The Napster model, due to its centralized design, is able to handle the two design challenges easily. In particular, every peer registered in the system knows the identity of the centralized index node. This index node has a directory of all objects currently available. But a centralized index does not scale very well and is a central point of failure.

A completely distributed approach is to have each peer build an index over its own files, and queries are flooded into the peer-to-peer network (e.g. Gnutella [4]). In this approach, a peer needs to maintain information only about its own data for the object lookups and information about its neighboring peers for request routing. A request from a peer is flooded via the neighboring peers. This approach has the advantage that peer nodes only maintain local information and the system does not have a central point of failure. However, flooding the network for every user query results in significant network and system overhead and hence the approach is not scalable.

Another approach that falls in between Napster and Gnutella is of building a superpeer network (e.g. KaZaA [8]) where smaller peers connect to a superpeer that builds an index over the objects shared by its set of peers. In addition to this, each superpeer keeps information about neighboring superpeers in the system. A peer can submit a query to its superpeer and the superpeer can lookup its index to see if another peer in its own territory can provide the object. Otherwise, it forwards the query to neighboring superpeers in the network. Recent literature on P2P systems [11] classifies these approaches as *unstructured* P2P systems.

Another interesting approach of indexing available data objects in the system, is to build a distributed hash table(DHT) [16, 14, 13]. Chord [14] hashes the objects into a ring formed by the peers in the system. The peers maintain routing information about other peers at logarithmically increasing distance in the ring. A querying peer hashes the name of requested object and then uses the routing information to forward the query to an appropriate peer. CAN [13] hashes the objects into a d-dimensional coordinate space, where parts of the space are owned by peers. The peers maintain routing information about the 2d neighbors in the

coordinate space. When a peer asks for an object, the object name is hashed and then the peer holding the desired hash key is located taking advantage of the structure of the hash space. The query is forwarded to this peer via neighbors and the peer can send the requested object to the querying peer. Advantages of these schemes are that they are completely distributed and highly scalable. Moreover they do not flood the network and direct the request toward a peer that holds the relevant information. These approaches are classified as *highly structured* P2P systems [11].

These P2P architectures, however, are confined to support file sharing applications over the Internet. The basic functionality that is supported by these P2P architectures is to provide exact-match query facility. Although the implementations are significantly different, all these systems support a hash-table interface of `put(key, value)` and `get(key)`. In general these systems are highly scalable; lookups can be resolved in $O(logN)$ or $O(dN^{\frac{1}{d}})$ for small d overlay routing hops for an overlay network with N peers. The fundamental limitation of these systems is that they only support *exact-match* lookups.

Since P2P systems provide scalable storage and efficient retrieval (at least for exact-match queries), database researchers have begun to ponder if P2P systems can be designed to provide complex query facilities on top of these DHT-based P2P systems. In particular, Gribble et al. [5] in their position paper provocatively titled "What can peer-to-peer do for databases, and vice versa?" outline some of the complexities that need to be overcome to gainfully exploit P2P systems for database query processing.

Similarly, Harren et al. [6] explore the issue of supporting complex queries in DHT-based P2P networks. Harren et al. report the implementation of database join operation over CAN [13], by performing a hash join of two relations R and S using DHT. They leave the question of developing range predicate selection over current DHTs as an open problem.

The work reported in this paper is similar in spirit to that of Harren et al. [6], in that we are interested in supporting database query processing over P2P networks. We specifically address the problem of executing a selection operation over a database relation using the information that is cached at different peers in the system. The main motivation for this is that the selection operation is typically involved at the leaves of the 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 desired data partition from the P2P system instead of fetching it from the base relation at the data source. Another motivation for our approach is that P2P users often ask broad queries even when they are only interested in a few results and therefore do not expect perfect answers [6].

In this paper, we present an architecture for a peer-to-peer system that shares data in the form of relational objects. In its simplest form peers can cache horizontal partitions of various relations. A peer can submit a query in the form of an SQL statement. The system tries to locate peers that have the most relevant partitions for the submitted query. We use a scheme based on Locality Sensitive Hashing [10, 7] to locate partitions of relations that are relevant to the query. Our main contributions are an architecture for a relational data sharing peer-to-peer system and a hashing based mechanism to locate data partitions relevant to a query. This paper constitutes an initial step to enable general query processing over P2P data sharing architectures.

2 Data Sharing in P2P Systems

We consider a system consisting of peers connected to each other via connections over a TCP/IP network. The peers form a data sharing system where the shared data is in the form of database tuples and relations. We assume a global schema that is known to all the peers in the system. Sources of data are part of the peer-to-peer system (i.e., they are also peers in the system), and are known to all the peers. However, access to the base relations may in general be undesirable due to load and connectivity reasons. In addition to the sources, other peers are allowed to cache horizontal partitions of relations. Peers are allowed to submit SQL queries to the system. We pose the following restriction on the queries: the selects on a relation can be only on one attribute at a time. The peer converts the query into a plan where all the selects are moved toward the leaves as much as possible. This is a well known algebraic optimization technique [15]. In such a plan, the peer can now request to locate relevant relation partitions in the system that can help answer the query. The located peers caching relevant partitions can send the data over to the requesting peer which can now compute the remaining query locally using the available data.

To illustrate the behavior of the system, let us consider the following example. Assume that the following relations exist in the global schema:
Patient(patient_id, name, age),
Diagnosis(patient_id, diagnosis, physician_id, prescription_id),
Physician(physician_id, name, age, specialization), and
Prescription(prescription_id, date, prescription, comments).
Suppose a peer wishes to find out what prescriptions have been provided to patients diagnosed with *Glaucoma* and with age in 30 to 50 between Jan 2000 and Dec 2002. More formally, in SQL the query can be written as:

Select Prescription.prescription
from Patient, Diagnosis, Prescription
where $30 \leq age \leq 50$
and diagnosis = "Glaucoma"
and Patient.patient_id = Diagnosis.patient_id
and $01-01-2000 \leq date \leq 12-31-2002$
and Diagnosis.prescription_id
= Prescription.prescription_id.

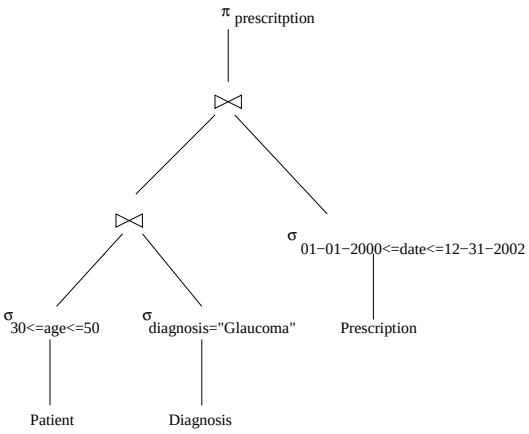


Figure 1: A possible query plan

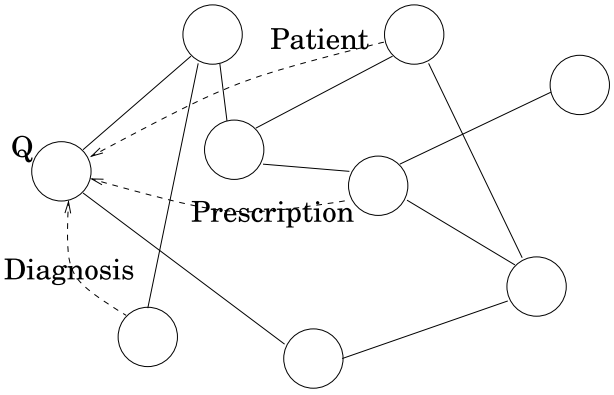


Figure 2: System Overview

A possible plan for the above query is shown in Figure 1. The peer can now ask the system to locate partitions of relations Patient, Diagnosis and Prescription which match the selection conditions. See Figure 2. The node labeled Q is the peer at which the query is initiated. It produces the above plan and hashes on the desired partitions to locate peers that can provide relevant data [1], i.e., Patient tuples with $30 \leq age \leq 50$, Diagnosis tuples for *Glaucoma* and Prescriptions with $01-01-2000 \leq date \leq 12-31-2002$. Once peer Q has obtained the data from the peers that have the desired partitions, it can now perform the join operations and project the desired attribute. Hence, in general, the problem of answering any SQL query depends critically on the efficient retrieval of selected partitions of the desired relations in the system. In this paper we focus on this problem, and in our future work we extend this approach to general queries.

[1] A query specifies a range over an attribute of a relation. We refer to the resulting set of tuples defined by this range as a *data partition*.

3 Locating Relevant Partitions

The problem of locating relevant data partition of a relation can be formulated as follows: Given a relation R and the selection range $(start, end)$ over an attribute of the relation, find out if there is a peer that caches a partition of relation R that can help us compute the desired selection.

3.1 Motivation

We start by considering the simpler problem of distributing and retrieving specific tuples of a relation with a given requested key. Any of the distributed hash tables(DHT), e.g., CAN [13] or Chord [14], can be used for this purpose. In particular, consider a query of the form:
*Select * from Patient where* $age = 30$.
In this case we consider the key to be *age* with value 30. The first time this query with this specific parameter setting ($age = 30$) is posed, the query must be routed to the source to retrieve the set of tuples in Patient with $age = 30$. Now we use a DHT to store this partition of Patient at a peer in the system. Subsequent queries with $age = 30$ would immediately map to that peer and hence would not need to overload the source site. This approach can be easily extended to support exact matches of ranges (or selection queries with predicates). Consider the following query:
*Select * from Patient where* $30 \leq age \leq 50$.
In this case, we could use the specific range $[30-50]$ as a key, which is used to hash the qualifying tuples. When a query is later posed with exactly the *age* range of $[30-50]$, this cached partition at a peer can be retrieved instead of going to the source relation.

This approach although simple, only supports key-range lookup for exact matches. However, even if the requested query partition $[start, end]$ does not exist there may be another partition $[start-\epsilon, end+\epsilon']$ which could have easily satisfied the query. In particular a query asking for all patients with ages between 30 and 49 would not hash to the same peer and hence would not benefit from the stored partition although this new query is very similar to the previous one. In fact the entire answer set is contained in the cached partition.

In a centralized system all the data partitions are at one location and the problem of finding a data partition that contains the query selection range can be solved by building an index over the stored data partition ranges. In a P2P system the data partitions are

distributed over various peers across a wide area network, and the problem becomes more complicated because in addition to finding the right data partition we also need to find where the partition is. Unfortunately the problem of discovering partitions in a P2P system that contain the selection range is extremely hard to solve exactly. In general, the problem of determining containment of a query in a given set of views is NP-complete [9]. Furthermore, P2P users often ask broad queries in order to find data of their interest and do not expect exact answers for their queries [6]. We therefore approach this problem by trying to develop techniques that provide approximate answers. Our approach is based on DHTs where *similar* ranges are hashed to the *same* peer with high probability and hence we can potentially benefit from previously cached partitions. Our solution is based on Locality Sensitive Hashing introduced by Motwani and Indyk [7] for the nearest neighbor problem. The existence of such hash functions was first shown by Linial and Sasson [10]. In the following we take a slightly different definition of locality sensitive hashing than [10]. We have adapted the definitions from [7, 2].

3.2 Locality Sensitive Hashing

If A, B are two sets of values from domain D then a family of hash functions H is said to be *locality preserving* if for all $h \in H$ we have:

$$Pr[h(A) = h(B)] = sim(A, B)$$

where $sim(A, B)$ is a measure of similarity of the sets A and B. If Q, R represent the range sets in the query and the matched answer respectively, then we would like to use a similarity measure defined by containment, i.e.,

$$sim(Q, R) = \frac{|Q \cap R|}{|Q|}.$$

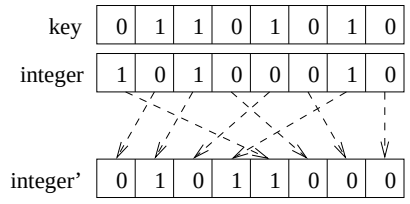
In [2], Charikar shows that if a similarity measure $sim(Q, R)$ admits a locality sensitive hash family then the corresponding distance function $\Delta(Q, R) = 1 - sim(Q, R)$ satisfies the triangle inequality:

$$\Delta(Q, R) + \Delta(R, S) \geq \Delta(Q, S)$$

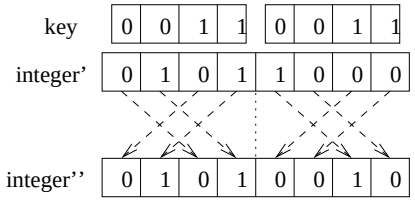
It turns out that the similarity measure based on containment of sets as defined above does not satisfy the triangle inequality. Hence no locality sensitive hash functions exist for the containment similarity measure. On the other hand, for the Jaccard set similarity measure

$$sim(Q, R) = \frac{|Q \cap R|}{|Q \cup R|}$$

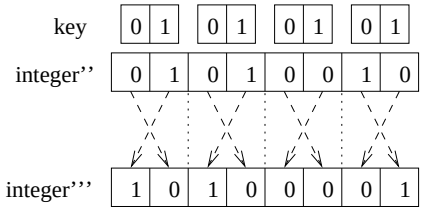
$1 - sim(Q, R)$ does satisfy the triangle inequality and there exists a locality sensitive hash function family for this similarity measure given by *min-wise independent permutations* [2, 1].



(a) first iteration



(b) next iteration



(c) last iteration

Figure 3: Iterations of the permutation operation

3.3 Min-wise Independent Permutations

The hashing scheme given by min-wise independent permutations is as follows. Given a domain D, consider a random permutation π of D. Assume that the elements of D are totally ordered. Given a range set $Q \subseteq D$, the hash function h_π is defined as:

$$h_\pi(Q) = min\{\pi(Q)\}$$

Then the property satisfied by this hash function family is that

$$Pr_\pi[h_\pi(Q) = h_\pi(R)] = \frac{|Q \cap R|}{|Q \cup R|}$$

which is the Jaccard set similarity measure.

For an integer in Q, the permutation operation is performed as described below. Assume we are dealing with sets of 8-bit integers. Take an 8-bit key that has exactly 4 random bits set to 1. For an integer in the set, move the bits corresponding to the position of 1's in the key to upper half and the others to the lower half in order. The operation is illustrated in Figure 3(a) for an 8-bit number and key.

Next, choose a 4-bit key with exactly 2 random bits set to 1. Again permute the bits in the 8-bit integer

using this key for each of the 4-bit halves. This is illustrated in Figure 3(b). And so on, until each pair of 2 bits has been permuted (see Figure 3(c)). The keys for this permutation function are representable as two 8-bit integers. The permutation operation produces $integer'''$ as the final output. The hash function $h_\pi(Q)$ applies the above permutation operation on each integer in the range Q and then takes the minimum of the resulting integers.

4 System Architecture

Given a selection operation for a relation we wish to locate the peers that have cached relation partitions that are a nearby match for the selection range of the given query. Instead of flooding the network with the query or going to various distributed indices as in a superpeer network, we want to use hashing to locate the peers that have relevant data partitions. So a distributed hash table needs to be maintained over the peers in the system. And since nearby matches of data partitions are to be located we use locality sensitive hashing to hash data partitions. Let us call the integer value produced by the hash function as *identifier*. An appropriate family of locality sensitive hash functions is used to map the data partitions to a 32-bit identifier space. These identifiers identify the buckets in the distributed hash table. Because of the property of locality sensitive hashing similar data partitions hash to nearby identifiers.

The next problem is how to store the distributed hash table over the peers in the system. Our general approach for locating relevant partitions for a given selection query can be summarized as follows:

1. The query range is hashed to the identifier space using an appropriate locality sensitive hash(LSH) family.

2. Peers in the system are also mapped to the same identifier space using any randomly distributed hash function (e.g. SHA-1 [3]).

3. We use Chord [14] to map data partition identifiers to peer node identifiers, and provide the lookup and routing facility.

Figure 4 illustrates how the distributed hash table is created. The identifier space (in this paper we will use a 32-bit space) is organized as a ring. The peer nodes are hashed using a hash function (such as SHA-1 [3]) over their IP address into the identifier space. The range specifying a data partition is also hashed into the same identifier space using locality sensitive hashing. From the properties of locality sensitive hashing(LSH) similar ranges are hashed to the same identifier with high probability. Since the domain of data partitions is much larger than the number of peer nodes in the system, we use Chord [14] to consistently map multiple data partitions to the same peer node. This mapping is based on Chord's circular structure. Each data partition identifier i is mapped to the peer node with the least identifier greater than or equal to i in the circular identifier space. A peer is thus responsible for all hash buckets corresponding to identifiers from the identifier of its predecessor node (excluding it) to itself.

To locate a given identifier, each peer in Chord also maintains information about other peers in the identifier ring that are at logarithmically increasing distances. Using this information, the peer holding a requested identifier can be located in $O(logN)$ lookups where N is the number of peers in the system. Once the peer is located, the bucket corresponding to the requested identifier is searched for the most similar range.

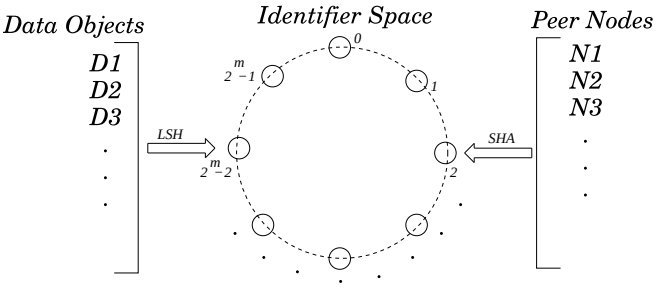


Figure 4: Structure of the Distributed Hash Table

We now discuss in more detail how data partitions corresponding to range selections are hashed into the identifier space. Given a selection query range Q we consider the range as the set of values from which the identifier is computed, e.g., for the range $30 \leq age \leq 50$, the set of values is $\{30, 31, 32, \ldots, 50\}$. Using a min-wise independent permutation hash function h_π, we derive the *identifier* for this set. From the properties of these hash functions we know that this identifier maps to a similar identifier (which corresponds to a range partition R) with probability $p = sim(Q, R)$, where $sim(Q, R)$ is the Jaccard set similarity measure.

Since, this is a probabilistic approach, we would like to hash similar ranges to the same identifier with high probability, and have a low probability of collision for dissimilar ranges. This could be done following an approach suggested in [7]. Consider a group $g = \{h_1, h_2, \ldots, h_k\}$ of k hash functions selected uniformly at random from the family of hash functions. Then the probability that the two sets hash to the same value for all k hash functions, i.e., $Pr[g(Q) = g(R)] = p^k$. Now, lets say we have l such groups $g_1, g_2, \ldots, g_l$ of hash functions. The probability that Q and R do not agree for a group g_i is $1 - p^k$. And the probability that they do not agree for all l groups is $(1 - p^k)^l$. So, the probability that Q and R agree on at least one of l groups is $1 - (1 - p^k)^l$. Therefore, if we use l groups each of k randomly selected hash functions we

can obtain l hash values for sets of tuples identified by their selection range and store them at peers that are responsible for the obtained hash values. Then depending on the values of parameters k and l, with high probability at least one of those l peers will have a data partition that contains data relevant to the desired range. Let Q denote the range of selection for the relation partition. Here is a rough sketch of the procedure followed.

```
At the querying peer:
For each g[l] do
 \\g[l] is a group of hash functions
  identifier[l] = 0;
  For each h[i] in g[l] do
    identifier[l] ^= h[i](Q);
  done
done
For each identifier[l] do
  Send a request to peer holding
   the identifier for the desired
   partition.
done
Get replies from all the peers.
Select the best match from all
 the replies you have got.
If none of the match is exact,
 also store the computed
 partition at the peers holding
 the computed identifiers.
```

In the above procedure, l identifiers are computed for a range set and peers holding those identifiers are contacted. There can be at most l different peers holding the identifiers. Each contacted peer checks the list of partitions that it has associated with the identifier and finds the best match for the query partition in the list and sends the best match to the requesting peer. The requesting peer can now choose the best match from the l replies it gets, and contact the peer with that partition for the data of the partition.

5 Experimental Results

In this section we analyze the performance of the proposed range selection in P2P systems in terms of the quality of matched partitions obtained and in terms of the scalability of the system.

5.1 Performance of Hash Functions

As shown in Section 4 we can use l groups of k hash functions to find matching partitions for a given query range with probability $1-(1-p^k)^l$, where p is the similarity of the queried range and the matched partition measured with Jaccard set similarity measure. For our experiments we chose the values for parameters k and l to be 20 and 5 respectively, because these values make the function $1-(1-p^k)^l$ to reasonably estimate a step function with a step at 0.9.

The min-wise independent permutations from Section 3.3 can be computationally expensive. Hence, we have also explored the family of *linear permutations* given by $\pi(x) = ax + b \ mod \ p, a \neq 0$ [1]. In addition to linear permutations, we also tried another family of *approximate min-wise independent permutations* which are just the first iteration (Figure. 3(a)) of the min-wise independent permutations. This approximate family is representable with a single 32-bit integer key and is computationally less expensive.

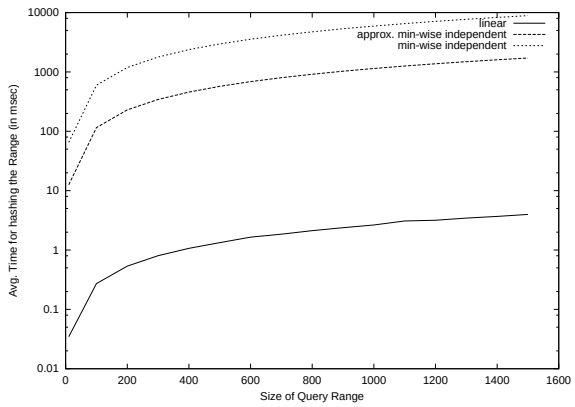
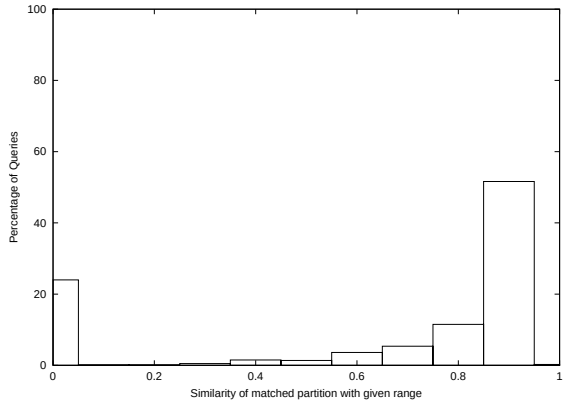


Figure 5: Execution times for the hash function families.

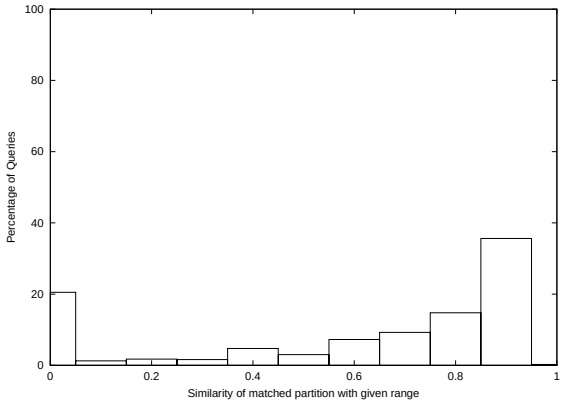
Figure 5 shows the execution times in milliseconds on a 900 MHz Pentium for the $l \times k$ (100) hash functions for query range sets with sizes varying from 10 to 1500. On the average, linear permutations are 1000 times faster than min-wise independent permutations. And, approximate min-wise independent permutations are about 10 times faster than min-wise independent permutations.

We used a set of 10,000 integer ranges with integers in 0 and 1000 as our query selection ranges. The range sets were generated uniformly at random and had only 0.2% repetitions. We start with an empty system and cache any query range if it is not already stored. We measured the similarity of the matched data partition for a given query range, where the similarity measure is the Jaccard set similarity measure. Figures 6 and 7 presents the results for the three hash function families described above with a warmup period of first 20% of the queries removed. The x-axis in the graphs represents the similarity between the range in the query partition and the range of the matching partition found. The similarity of two data partitions is 0 if they do not have any tuples in common, and it is 1.0 if they are identical. The y-axis is the percentage of total queried partitions that found a match with the given similarity measure.

From Figure 6(a) we can see that almost 50% of the queried partitions found a matching partition



(a) Min-wise Independent Permutations



(b) Approximate Min-wise Independent Permutations

Figure 6: Performance of hash functions

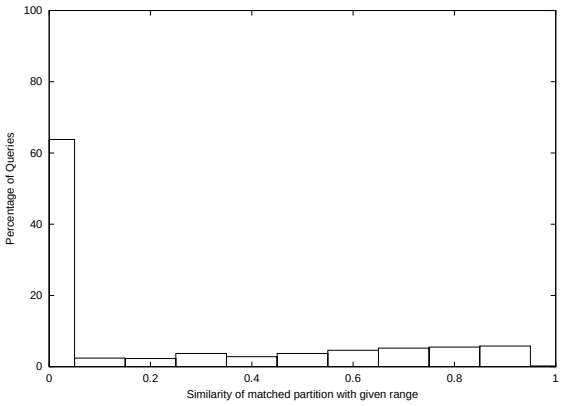


Figure 7: Performance of Linear Permutations

with similarity between 0.9 and 1.0. The queried ranges contain only 0.2% repetitions, hence the identical matches are very low. But as we see there are almost 25% of the queries that did not find good matches at all. This is so because min-wise independent permutations are *very good* hash functions and they try to imitate the ideal step function with a step at similarity 0.9. Therefore, for quite a few queries they do not find matches that may exist but are not as similar as 0.9. Furthermore, since the query ranges are uniformly distributed there are always new query ranges for which there are no stored similar data partitions.

Figure 6(b) shows the quality of matches obtained by approximate min-wise independent permutations. Approximate min-wise independent permutations find good matches for almost 35% of the queries. Unlike min-wise independent permutations though, they do try to find matches for queries even though the matches may not be as good as 0.9. Also, they are much faster to compute than min-wise independent permutations because they only need to perform the first iteration of the complete permutation operation as described in Section 3.3.

Linear permutations are easily representable and very efficiently computable. As the graph in Figure 7 shows the quality of matches obtained by them is not good. Although, they do find an identical match if it exists. As the system evolves, the probability that identical queries had been asked earlier goes higher and linear permutations will tend to produce better results.

5.2 Recall

The previous section measured the performance of hash function families in terms of Jaccard similarity measure, but from a user perspective we are more interested in how useful the match is for answering the query. A measure of the usefulness of the matched data partition is *recall*, which is a measure of how much of the desired answer is given by the matched partition. Figure 8 shows the recall of matched data partitions for the three hash function families. The x-axis in the graph is the portion of the desired answer obtained by the matched partition. The y-axis is the percentage of queries that are answered upto a given portion.

The min-wise independent permutations are able to answer almost 30% of the queries completely. The approximate min-wise independent permutations answer about 35% of the queries completely. Linear permutations answer 50% of the queries completely. Approximate min-wise independent permutations and linear permutations lead to better containment results than min-wise independent permutations because they are not too strict about finding data partitions that are similar. Hence, they match broader partitions which contain more of the desired answer. However, this looser matching property of linear permutations results in poorer recall quality for rest of the queries when compared to the other two hash families. In general, min-wise independent permutations and approxi-

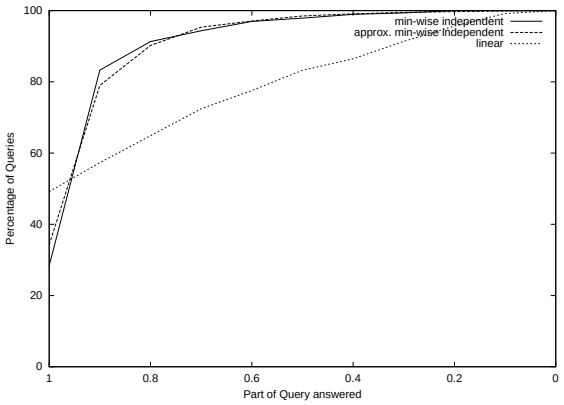


Figure 8: Recall for the hash function families.

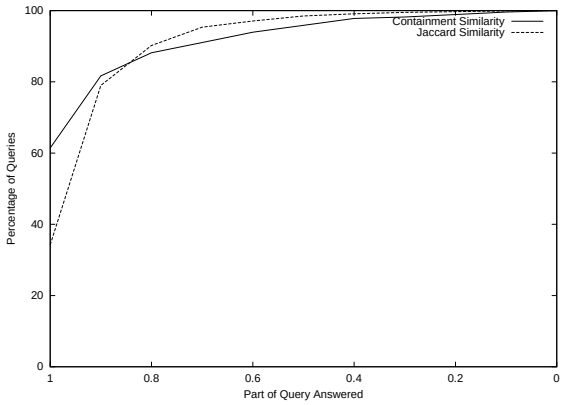


Figure 9: Recall with Containment Similarity matching

mate min-wise independent permutations have similar recall. They answer at least 0.8 of 90% of the queries, and about 98% of the queries get at least half of their answers.

As we saw earlier in Section 3.2, we cannot use the containment similarity measure

$$sim(Q, R) = \frac{|Q \cap R|}{|Q|}$$

to define our locality sensitive hash function families because it does not satisfy the triangle inequality. But once we have hashed a selection range to an identifier using a hash family defined according to the Jaccard set similarity measure, we can use containment similarity to find the best match from the hash bucket. Figure 9 shows the recall when we use containment matching in conjunction with approximate min-wise independent permutations. Both of the schemes use approximate min-wise independent permutations for hashing the selection ranges. Using the containment similarity measure the percentage of queries completely answered improves from approximately 35% to almost 60% of the queries, and for approximately 85% of the queries the recall is better with the more realistic similarity measure. However, for the remaining 15% the mismatch between the principle the hashing is based on and the actual measure used shows in improved performance for the Jaccard similarity measure.

Since P2P users often ask broad queries and do not expect exact answers [6], the system can present the user the part of the answer it is able to find fast, and can also let them know what selection ranges this answer corresponds to. If the user is not satisfied with the answer, they have a choice to go to the source for the rest of the answer. We have also explored the option of submitting a padded query. Instead of going to the source, the system evaluates the user query with its selection ranges expanded. Figure 10 shows the recall when the selection ranges are expanded 20% on the edges and approximate min-wise independent permutations are used for the hash functions. With query padding a little over 70% of the queries are answered completely. This represents a doubling of the queries completely answered when compared with no padding with Jaccard similarity measure (compare with Figure 9). However, padding does have a cost. Although approximately 78% of the queries benefit and show improved performance over no padding, for the rest of the queries, the extended range results in lesser recall than without padding. This shows that there is a tradeoff between getting complete containment for queries versus total recall for all the queries. Padding is beneficial for the former whereas no padding for the latter. In future, we will explore dynamically adjusting padding for better overall performance.

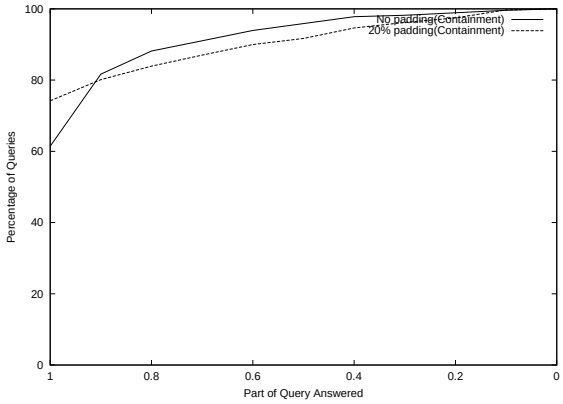
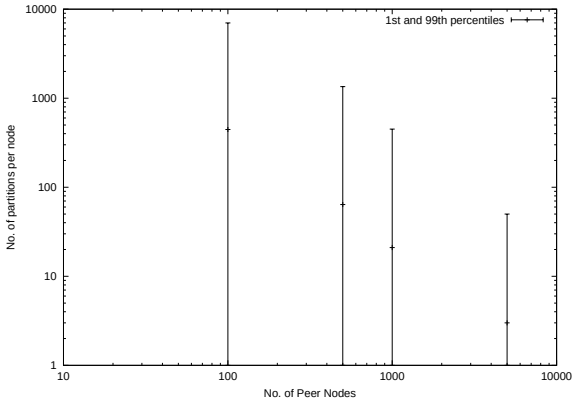


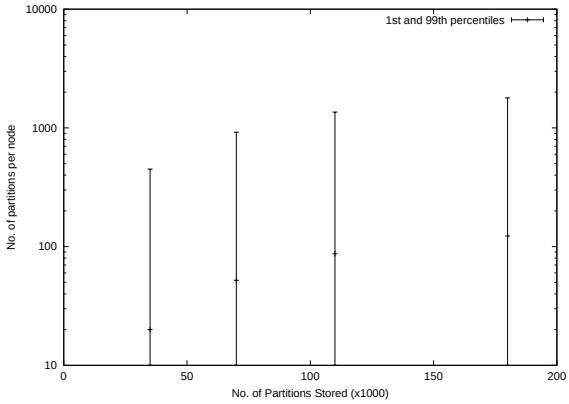
Figure 10: Recall with 20% Query Padding

5.3 System Scalability

We analyze the scalability of the system using simulation of the distributed P2P system. For the purpose of our simulation experiments we have modified the simulator of Chord [14] in the following manner. The `find` operations in our simulations take a query range set instead of a document id and hash the range set to 5 32-bit identifiers using the approximate min-wise independent permutations. The peer nodes holding the identifiers are discovered using the Chord lookup algorithm [14]. These peers then look up the corresponding

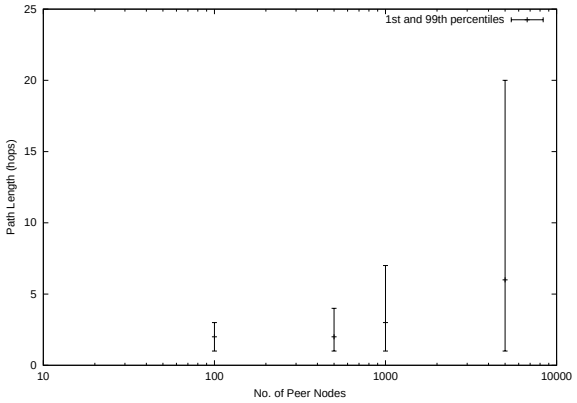


(a) Load distribution when the system stores 50,000 partitions.

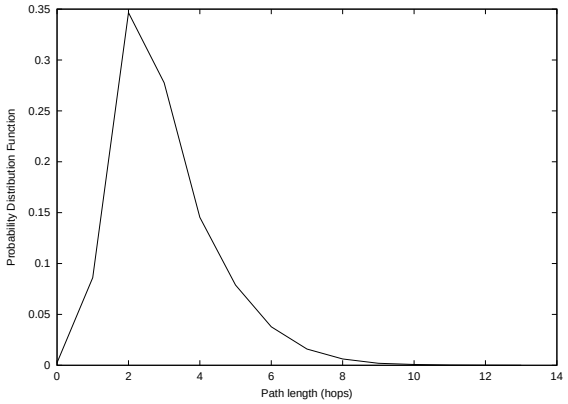


(b) Load distribution in a 1000 node system.

Figure 11: Load balancing in the system.



(a) Path length as a function of No. of nodes



(b) Prob. Distribution Function of path length in a 1000 node network

Figure 12: Path lengths for lookup operations

buckets for the identifiers they hold and find the best match for the query range set using the Jaccard set similarity measure. If the match is not exact, the new query range is also stored at those peers. There are no explicit `insert` operations in our simulation and the system starts with no partitions stored.

For analyzing the scalability of the system as the number of peer nodes grow we consider a system that stores 5×10^4 partitions. There are 10^4 unique partitions, and each is stored with five different identifiers computed by five different sets of hash functions. The number of peers in the system vary from 100 to 5000. Figure 11(a) shows the mean and the 1st and 99th percentiles of the number of partitions stored per node in the system as the number of peer nodes in the system increases. The load distribution gets linearly better with the increase in the number of peers. Figure 11(b) shows the mean and the 1st and 99th percentiles of the number of partitions stored per node in a 1000 node system where the number of total partitions stored in the system varies from 35000 to 180000. The mean values for the load distribution grow superlinearly with the increasing number of partitions but the 99th percentiles show a sublinear increase.

Path length is a measure of the number of hops in the overlay network that are required to route a query to the destination peer. We ran simulations with varying number of peers in the system and storing 5×10^4 partitions. Figure 12(a) shows the mean and 1st and 99th percentiles of path lengths for systems with number of peers varying from 100 to 5000. The mean path lengths are of the order $\frac{1}{2} \log N$ where N is the number of peers in the system. Figure 12(b) shows the probability distribution function of path lengths

in a 1000 node network. For most lookups with high probability the path length is 2. In general, the results are consistent with the results for exact-match queries [14].

Locality sensitive hashing hashes similar ranges to nearby identifiers. In the ring structure formed by the peers in Chord all identifier buckets falling between two peers get stored at the successor peer [14]. Potentially, we could now build up an index over all the partitions that get stored in various buckets at a peer. When we need to find a similar match for a query selection range, we can search through this index after locating the peer that holds the identifier for the selection range instead of just looking at the ranges in the bucket of the identifier. Interestingly, with this approach the recall will be best when there is just one peer in the system, since all partitions will be stored at that peer, and for a lookup we would be searching through the index at that peer. As the number of peers in the system grows, the partitions will get distributed at the peers and for each search we would be looking at a smaller index. But in the worst case, a peer would hold buckets for at most one identifier, in which case we would look through the partitions in only that bucket, and the recall would still be as good as the results presented in Section 5.2.

6 Conclusions and Future Work

We have presented an architecture for a peer-to-peer data sharing system that shares data in the form of database relations. Peers in the system cache horizontal partitions of the relations based on the queries executing in the system. We have also presented a novel approach for locating relevant data partitions in the peer-to-peer system using locality sensitive hashing. The benefit of such an approach is that it not only finds exact partitions in the system if they exist, but also can help locate partitions that nearly match the ones required by the query.

In the future, we will address the problem of locating horizontal partitions obtained by multiattribute selections. We would also like to investigate caching general query results in the system in addition to horizontal partitions of relations. Furthermore, the problem of planning a query in a peer-to-peer system based on available statistics of the system is worth exploring.

Acknowledgments

We would like to thank Chord research group at MIT for providing us the Chord simulator for conducting experiments. The work of Abhishek Gupta was supported by an IBM Cooperative Fellowship.

References

[1] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 327–336. ACM Press, 1998.

[2] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 380–388. ACM Press, 2002.

[3] FIPS180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, April 1995.

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

[5] Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan Suciu. What can peer-to-peer do for databases, and vice versa? In *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB 2001)*, Santa Barbara, California, USA, May 2001.

[6] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Than Loo, Scott Shenker, and Ion Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of the first International Workshop on Peer-to-Peer Systems*, 2002.

[7] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM Press, 1998.

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

[9] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 95–104. ACM Press, 1995.

[10] Nathan Linial and Ori Sasson. Non-expansive hashing. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 509–518. ACM Press, 1996.

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

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

[13] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[14] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[15] J D Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

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