Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks

Peter Triantafillou, Theoni Pitoura

[1] Department of Computer Engineering and Informatics,
University of Patras, Greece
{peter, tpit}@ceid.upatras.gr

Abstract. In this work we study how to process complex queries in DHT-based Peer-to-Peer (P2P) data networks. Queries are made over tuples and relations and are expressed in a query language, such as SQL. We describe existing research approaches for query processing in P2P systems, we suggest improvements and enhancements, and propose a unifying framework that consists of a modified DHT architecture, data placement and search algorithms, and provides efficient support for processing a variety of query types, including queries with one or more attributes, queries with selection operators (involving equality and range queries), and queries with join operators. To our knowledge, this is the first work that puts forth a framework providing support for all these query types.

1 Introduction

Recently, P2P architectures that are based on Distributed Hash Tables (DHTs) have been proposed and have since become very popular, influencing research in Peer-to-Peer (P2P) systems significantly. DHT-based systems provide efficient processing of the routing/location operations that, given a query for a document id, they locate (route the query to) the peer node that stores this document. Thus, they provide support for exact-match queries. To do so, they rely, in general, on lookups of a distributed hash table, which creates a structure in the system emerging by the way that peers define their neighbors. For this reason, they are referred to as structured P2P systems, as opposed to systems like Gnutella[1], MojoNation[2], etc, where there is no such structure and, instead, neighbors of peers are defined in rather ad hoc ways.

There are several P2P DHTs architectures (Chord[3], CAN[4], Pastry[5], Tapestry[6], etc.). From these, CAN and Chord are the most commonly used as a substrate upon which to develop higher layers supporting more elaborate queries.

CAN ([4]) 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 data 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 data object is mapped to a point in d-dimensional space and then the request is routed towards 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 for exact match lookup is O($dN^{1/d}$) hops.

Chord ([3]) hashes both the key of a data object into an *m*-bit identifier, and a peer's IP address into an *m*-bit identifier, for a value of m that is large enough to make the probability of two peers or keys hashing to the same identifier negligible. Identifiers are ordered in an identifier circle modulo 2^m, called the Chord ring. The keys' identifiers map to peers' identifiers using consistent hashing (i.e. a function succ(k), for k being the identifier for the data object's key). The peers maintain routing information about other peers at logarithmically increasing distance in the Chord ring (i.e. in tables stored in each peer, called finger tables). Therefore, when an exact match query for a data object is submitted in a Chord P2P architecture, the querying peer hashes the attribute's value, locates the peer that stores the object, and uses the routing information stored in the finger tables to forward the query to this peer. As a result, Chord requires O($logN$) routing hops for location/routing operations.

There are also other than DHTs structured P2P systems, which build distributed, scalable indexing structures to route search requests, such as P-Grid. P-Grid ([7]) is a scalable access structure based on a virtual distributed search tree. It uses randomized techniques to create and maintain the structure in order to provide complete decentralization.

Although P2P networks provide only exact-match query capability, researchers in data management have recently began to investigate how they could enhance P2P networks to reply to more complex queries. This led to research addressing specific query types (such as range queries, queries involving join, and/or aggregation operators), etc. However, a proposal of a framework upon which all complex query types could be efficiently supported in a P2P architecture is very much lacking.

With this work we attempt to present a unifying framework for supporting efficiently complex querying in P2P networks. Given the prominence of DHT-based architectures, we will assume a DHT architecture as our substrate, hoping that in this way our work can leverage existing research results to develop a comprehensive query processing system for P2P networks.

2 Related Work

Gribble et al. ([8]) investigate how P2P systems can gain the strengths of data management systems in semantics, data transformation and data relationships, and therefore, enable them to provide complex queries capabilities. Towards that direction, they define the dynamic data placement problem in P2P systems and propose a decentralized, globally distributed P2P query processor, which would give answers to aggregations, range, and more complex queries in P2P systems. However, in this work they do not provide specific solutions to query processing since their query processor is still in an initial phase.

In the following sections, we briefly present existing work in addressing specific query types, such as range queries, queries involving join, and/or aggregation operators, etc.

Supporting Range Queries

Gupta et. al [9] propose an architecture for relational data shared in a P2P system based on Chord, and a hashing-based method to provide *approximate* answers to range queries. Specifically, given a range, this approach considers the set of values that the range consists of, computes an integer for each one of the values by applying a min-wise independent permutation hash function ([10]) on each value, and finally takes the minimum of the resulting integers, as the identifier for that range. The min-wise independent permutations have the property to hash similar ranges to the same identifier with high probability (locality sensitive hashing). Since they use Chord to map ranges to peers, and to reply to range queries, lookup is performed in O(logN) hops, where N is the number of peers.

Sahin, et al. ([11]) extend the CAN DHT system ([4]) for d=2. The virtual hash space for a single attribute is a 2-dimensional square bounded by the lower and higher values of the attribute's domain. The space is further partitioned into zones, and each zone is assigned to a peer. That peer stores the results of the range queries whose ranges hashes into the zone it owns, as well as routing information about its neighbors - owners of its adjacent zones. When a range query is submitted, it is routed towards the range's zone through the virtual space. Once the query reaches that zone, the stored results at this zone are checked. If the results are found locally, they are returned. Otherwise, the query is forwarded to the left and top neighbors that may contain a potential result (recursively). The basic idea of this forwarding is that smaller sets of tuples that are answers to the original range query may be cached at peers that contain supersets of the given range.

Andrzejak and Xu ([12]) also propose a CAN-based extension for range querying, in grid information infrastructures. A subset of the grid servers is responsible for subintervals of the attribute's domain, the *interval keepers* (IK). Each server reports its current attribute value to the appropriate IK. Additionally, each IK owns a zone in the logical d-dimensional space. For efficient range queries, the Space Filling Curves, and especially the Hilbert curve ([13]) for R^2 mapping and its generalization for R^d are used to map the intervals to the zones (the Hilbert curve has the property of proximity preservation). When a range query is issued, the problem is how to route the query to the IK whose interval intersects the query range. There are many techniques to perform efficient routing for range queries, the best of which is propagating the query in two waves: the first one to the neighbors that intersect the query and have higher interval than the current node, and the second wave to those that have lower interval.

Supporting Multi-attribute Queries

To our knowledge, there is no efficient specific solution to this problem. Researchers either leave this issue to address it in the future, or address it by using separate instances of DHTs for each one of the attributes. However, such a solution is not efficient since it requires many replicas of same data objects in different peers, one for each of the index attributes.

An interesting work is presented in Felber et. al ([14]), although they do not aim at answering complex database-like queries, but rather at providing practical techniques for searching data using more advanced tools than keyword lookups. They create multiple indexes, organized hierarchically, which permit users to access data in many different ways (query-to-query mapping). Given a broad query, the system recursively queries itself until it finds the desired data items, which are stored on only one or few of the nodes. The good property of this system is that it does not replicate data at multiple locations, but it provides a key-to-key service, by storing at each node only the indexes to other nodes, which store the answers to more specific queries.

Supporting Queries Involving Joins

Harren et. al ([15]) propose a three-tier architecture: data storage, (any) DHT overlay network, and a Query Processor on top. Specifically, they partition the DHT identifier space into *namespaces* (relations). They also build a hierarchical identifier space on top of the flat identifier space, by partitioning the identifiers in multiple fields, in order to manage multiple data structures. *Joining* relations *R* and *S* is implemented by using multicast to distribute the query to all peers in the DHT network. The algorithm locally scans tables *R* and *S*, and republishes the resulted tuples in a temporary namespace using the join attributes as its identifier (i.e. it re-hashes the tuples in the DHT). Mini-joins are performed locally as new tuples arrive (i.e. they perform pipelined hash join), and the results are forwarded to the requestor. The same work addresses also projection, join and group by (aggregations) operations in a similar way.

3 The Proposed Framework

3.1 The Problem

We assume a P2P network, with nodes/peers publishing, storing, and sharing data. In our framework, data is organized in relations and each data object is described by a tuple with a number of attributes. For example, in a file sharing P2P system, files are described by their metadata, such as name, type, date, author, etc. forming tuples.

Our study starts from simpler queries, which we call *rudimentary* queries, and proceeds to more complex queries. To facilitate this we have defined a number of *query types*, for both the categories, according to the involvement of different parameters in the expression of the query, such as:

a. One or more attributes, forming single or multi-attribute queries, coined *SA or MA,* respectively
b. One or more conditions (single or multi-condition queries, respectively), using conditional operators (i.e. union (OR), intersection (AND))
c. Different operators applied on the attributes, such as equality and range operators
d. Special functions applied on the data, such as aggregations (i.e. count, sum, average, maximum/minimum, etc), grouping, ordering, etc.
e. Single or multi-relation queries coined, *SR, or MR,* respectively.

3.2 The Approach

We will classify expected queries into "*rudimentary*" and "*non-rudimentary*" queries, and define *query types* for each of the above categories. We will analyze how to support each query type, starting from rudimentary queries. The non-rudimentary query types that are combining features from these rudimentary types will be supported using the corresponding mechanisms for each involved rudimentary query type.

Based on the above we have classified the expected queries into the following query types:

Rudimentary Queries

[SR, SA, =]:	referring to a query over a single-relation, single-attribute, single-condition with an equality operator.
[SR, SA, < >]:	referring to a query as above, but with a range operator.
[MR, MA, join]:	referring to a query over multiple joined relations

Non-rudimentary Queries

[SR, MA, =]:	referring to a query over multiple attributes with equality operators.
[SR, MA, <>]:	referring to a query over multiple attributes with range operators.
[MR, MA, =]:	referring to a query over multiple relations, over multiple attributes with equality operators.
[MR, MA, <>]:	referring to a query over multiple relations over multiple attributes with range operators.
[MR, MA, =, sf]:	referring to a query over multiple relations over multiple attributes, with *special functions, such* aggregate functions (sum, count, avg, min, max, etc), grouping, ordering, etc.

We currently address only integers as the attribute's domain, and leave other data types, such as strings, dates, Boolean for future work (an approach for strict substring searches using n-grams is presented in [15]).

In the following sections we examine each of the above query types separately aiming to find a single, efficient solution for all query types. As far as we know, there is no single approach that offers a 'global' solution, supporting all query types above. Our goal is to provide this and set a framework for further improvements and extensions for higher-level query languages (i.e. semantically enriched query languages, such as XML).

In the proposed framework we will use Chord, because of its simplicity, its popularity within the P2P community (which has led to it being used in a number of P2P systems), but also because of its good performance (logarithmic in terms of routing hops and routing state) and robustness.

4 The Proposed Architecture

4.1 The Infrastructure

First, we assume that data stored in the P2P network is structured in only one relation *R (DA_1, DA_2 , .. DA_k)*, where k is the number of attributes, DA_i the attribute domains, and A_i the name of the attributes, for each $i \in \{1, 2, .. k\}$. For simplicity, we assume that all k attributes are used to index data (i.e. we ignore attributes that are not used for indexing, since they do not affect the proposed architecture; we assume that these can be stored in the tuples and recalled each time we access the tuples).

Each tuple *(a_1, a_2 , .. a_k)* in R, with $a_i \in DA_i$, for each $i \in \{1, 2, .. k\}$, holds a primary key, or simply *key*, which is a distinct identifier. This key can be either one of the attributes of the tuple, or can be calculated separately based on one or more of the attributes.

Based on Chord, our protocol associates with each peer P, from the set $\{0, 1, .. 2^m-1\}$, an m-bit identifier, *n*, using a base hash function, such as SHA-1([16]), on the peer's IP address. Similarly, it associates with each tuple (a_1, a_2 , .. a_k) in R with key *t*, an m-bit identifier from the same set $\{0, 1, .. 2^m-1\}$, using a similar hash function on *t*. The identifiers, which are elements of the set $\{0, 1, .. 2^m-1\}$, are ordered in an identifier circle modulo 2^m, i.e. the Chord ring[1], which is also referred to as the DHT identifier space.

Furthermore, we use consistent hashing to map tuples to peers, and specifically, we use the function *succ(): $\{0, 1, .. 2^m-1\}$ ® $\{0, 1, .. 2^m-1\}$*, similarly to Chord. Each tuple (a_1, a_2 , .. a_k) with key t is routed and stored on the first peer whose identifier is equal to or follows *t* in the identifier space. This peer is called the successor peer of identifier t and is denoted by succ(t). Each peer *n* maintains routing information in the following structures:

a) *finger table*, where each entry l is the first peer that succeeds n by at least 2^{l-1} on the ring (mod 2^m), for $1 \leq l \leq m$, and ,
b) link to its *predecessor* peer.

Given this setup, the system is able to reply to queries of type [SR, SA, =], when the queried attribute is the key of a tuple.

4.2 The Enhancements to the Infrastructure

In this section, we describe how to enhance the Chord architecture to enable it to support the other rudimentary query types for *any attribute*. To do this, we proceed as follows: for each tuple and for one of its, say, *k* (index) attributes, we hash the attribute value of the tuple and insert it to a peer in the DHT identifier space, using the Chord data insertion protocol and a special hash function (which we will define below).

In order to support range queries for a single-attribute, we should somehow be able to keep ordering information for the values of that attribute. A simple solution to this

[1] From this point forward, when we refer to a peer n, we consider that n is its identifier, and for a key t of a tuple that t is its key identifier (otherwise, it is mentioned specifically).

is to use *order-preserving (i.e. monotonic) hash* functions to hash the attribute values to the identifier space. Then, tuples at the Chord ring are stored in an ordered form (say, ascending) by the values of this attribute.

For every *i Î {1, 2, .. k}* we define an *order-preserving hash function* h_i*: DA*$_i$ ® *{0, 1, ..* 2^m*-1},* such that:

$$\text{value}_1 \leq \text{value}_2 \Rightarrow h_i\,(\text{value}_1) \leq h_i\,(\text{value}_2),\ \text{ for any value}_1, \text{value}_2 \in DA_i. \quad (1)$$

There are many order-preserving hash functions we could use, however, it is quite improbable that they also provide *randomness* and *uniformity*.

Here, we use k order-preserving hash functions, one for each (index) attribute, as follows. Let us say that DA_i = <low_i, $high_i$> for an attribute A_i and we assume that 2^m > $|DA_i|$. Then, we partition the identifier space into 2^m / s_i ranges, each of size s_i, such that:

$$s_i = 2^m / \left(high_i - low_i + 1\right) \quad (2)$$

Then, for every value $a_i \in DA_i$, we define,

$$h_i : DA_i \rightarrow \{0, 1, \dots 2^m - 1\}, \quad h_i(a_i) = \left\lceil \left(a_i - low_i\right) \times s_i \right\rceil \quad (3)$$

Now we turn to presenting the basic algorithm for data (tuple) insertion. A node join/leave algorithm is accommodated by executing the join/leave protocol of Chord.

Data Insertion

A tuple (a_1, a_2, .. a_k) with key t is inserted in the network, using Chord's consistent hashing function. We store the tuple in the peer succ(t).

In addition, we apply order-preserving hashing over each one of its k (index) attribute values, a_1, a_2, .. a_k, and find the m-bit identifiers $h_1(a_1)$, $h_2(a_2)$, .. $h_k(a_k)$. We apply succ() to each one of the k hashed values and store the tuple in the peers with identifiers succ($h_i(a_i)$), for each i ∈ {1, 2, .. k}.

Therefore, for each tuple inserted in the network, we store (k+1) maximum replicas of this tuple: one copy of the tuple with consistent hashing over its key, and k replicas distributed in the peers in an order-preserving form based on its k attributes.

To maintain k more replicas of each tuple may be inefficient, as far as storage requirements are concerned and requires more processing time for data updates to keep replicas mutually consistent, as well as more overhead when nodes join/leave the network. On the other hand, it is indisputable that replicas of data in a DHT network are required given the frequent topology changes, if data availability is to remain at reasonable levels. Furthermore, this approach keeps routing hops for lookups significantly low, since range and join queries can be processed efficiently – as we will describe below.

A way to overcome the increased replica-related overhead is to allow only one copy of a tuple to be stored; specifically, the one stored using consistent hashing on the tuple's key, t, i.e. the tuple stored at the succ(t) peer. The other k peers, whose id is produced by order-preserving hashing for each one of the k attributes, could only store the value of the attribute hashed (needed to compare during lookup) and a link to the succ(t) peer. In this way however, we would increase the peers' state, by adding to the finger table and the predecessor more pointers to peers, indexed by the values of their k attributes. This alternative architecture would also increase lookup and join processing, as we describe below.

5 Processing Query Types

5.1 Rudimentary Query Types

As described above, there are three rudimentary types, and their processing is described below.

Query Type [SR, SA, =]

Processing of this query type is based on the Chord lookup algorithm, utilizing the finger table entries of peers, which are maintained as the Chord protocol prescribes. In the pseudocode below *n* is the requestor.

```
INPUT: $a_i \in DA_i$, for i ∈ {1, 2, .. k}
OUTPUT: a list of tuples, with value $a_i$ of attribute $A_i$
BEGIN
calculate $h_i(a_i)$,using the order-preserving hash
         function $h_i$
n_target = Chord_lookup $h_i(a_i)$ on peer n
request–and–receive from n_target  the desired tuples
END
```

The data store at peer n_target is locally searched for those tuples of relation R for which $A_i = a_i$.

Since processing of this query type is based solely on the Chord lookup algorithm, it follows that the query type is resolved in O(logN) routing hops. (A proof of this is fairly straightforward and is left out of this paper).

Query Type [SR, SA, <>]

Similarly, given a range query with a range (*low*, *high*) of an attribute, we hash using the order-preserving hash function used for that attribute over *low* and *high*, and find peers succ(h_i(low)) and succ(h_i (high)). Because of the order-preserving hashing, the requested tuples are stored in the peers from succ(h_i (low)) to succ(h_i (high)), and are retrieved by following the successor links within the Chord ring. Specifically, at first the query is forwarded from *n* to succ(h_i(low)), which forwards the query to its successor, and this is repeated, till the query reaches the peer with identifier succ(h_i (high)).

```
INPUT: (low, high) $\subseteq DA_i$, for i ∈ {1, 2, .. k}
```

```
OUTPUT: a list of tuples, with values of attribute $A_i$
         falling within the range (low, high)
BEGIN
calculcate $h_i$(low) and $h_i$(high),using the order-
         preserving hash function $h_i$
n_start = Chord_lookup $h_i$(low) on peer n
n_end = Chord_lookup $h_i$(high) on peer n
forward (query, n, n_end) to n_start
receive tuples
END

Forward (query, n, n_end) executed on node $n_j$
INPUT (low, high) $\subseteq$ $DA_i$, for i $\in$ {1, 2, .. k}
OUTPUT: a list of tuples, with values of attribute $A_i$
         falling within the range (low, high)
BEGIN
local_find(R, $A_i$, $a_i$), with $a_i \in$ (low, high)
send matching tuples to node n
if $n_j$ != n_end
         forward (query, n, n_end) to its successor
END
```

The above algorithm calls the Chord lookup algorithm to find the peer storing the value *low* of the queried attribute, therefore it needs O(logN) routing hops for this task. Subsequently, the query will be forwarded to all nodes falling between succ(h_i(high)) and succ(h_i(low)), successively, following each time the successor link of the current node (it is a direct link and, therefore it needs O(1) hop each time). Therefore, forwarding needs as many routing hops as the number of nodes lying between succ(h_i(high)) and succ(h_i(low)), which may be from O(1) to O(N)! Therefore, the above algorithm needs O(N) routing hops in the worst case and O(logN) hops in the best. This worst case performance is of course undesirable and resembles performance of lookup algorithms in unstructured P2P networks.

For this reason we enhance our architecture using a number of special peers, called *range guards*.

Enhancing the Architecture using Range Guards

We select a number of peers, called range guards, *RG*, to keep replicas of all tuples whose values of a specific attribute fall in a specific range. Each RG also maintains routing information for their neighbor RG, i.e for the RGs responsible for the succeeding range. Additionally, there are direct links in each peer to its corresponding RG.

Specifically, we partition the domain DA of an attribute A into *l* partitions (buckets). To achieve load balancing distribution among the RGs, the partitions should be such that each RG keeps an equal number of tuples. Since this is application-dependent and hard to accomplish, we partition DA into *l* continuous and disjoint ranges of *size s* = *|DA| / l* (|DA| is the size of domain DA). This means that we divide the attribute domain into *l* equal-sized subranges - equal to the number of partitions desired, and therefore, *l* RGs are defined.

We repeat partitioning for each one of the attributes over which we expect range queries to be common.

Range Queries using Range Guards

A range query is now forwarded into the Range Guards, the ones whose union of their ranges is the *minimum* range (L, H) such that: (low, high) $\subseteq$ (L, H).

As above, given a range (low, high) of an attribute, we hash using an order-preserving hash function over *low* to find the peer succ(h_i(low)). We then forward the query to its corresponding RG, which locally looks up for the requested tuples and sends the matched ones back to the requestor. Then, we compare high≤H. If it is true, the algorithm ends, since all the requested tuples are lying in the RGs that have been already accessed. Otherwise, the RG forwards the query to its successor RG, which repeats the same process.

Therefore, this algorithm requires O(logN) routing hops to access succ(h_i(low), and O(*l*) routing hops to forward the query from one RG to the next until the query results have been completely gathered. Since l << N, it is obvious that the algorithm is significantly more efficient that the one presented before. Specifically, if we set *l*=logN, we have log(N) RGs in the network, and a range query would need O(logN) + O(logN) = O(logN) routing hops. Alternatively, we can set $l = \sqrt{N}$ and then the load on each RG will be significantly smaller, while still obtain much better performance and specifically O($\sqrt{N}$).

On the other hand, this enhancement requires more storage for routing information in a peer, since every peer should also keep a link to its corresponding RG. Additionally, we require the existence of *l* range guard peers with enhanced processing and storage capacity, able to store and process requests for a significant proportion of our data. However, in peer-to-peer data management settings, peers will be able to store complete database systems which implies a willingness and ability to store large amounts of data. Furthermore, this is not unrealistic, since many peers in the network have been proven to be altruistic and/or more powerful, willing and capable to offer their resources and thus play the role of RGs. In fact, a hot trend within the networking and distributed systems P2P research community is to exploit the heterogeneity with respect to the capabilities of peers (bandwidth, processing power, storage) in order to improve routing efficiency ([17]). With the notion and exploitation of range guards we can harness this power heterogeneity of peers in order to facilitate the efficient processing of range queries.

Additionally, with this extended architecture, there are more replicas of the tuples stored in the network. To avoid this degree of data redundancy, we could follow the alternative discussed earlier, and keep only a link to the succ(t), as well as a link to its corresponding RG. This alternative still only costs O(*1*) routing hops.

Query Type [MR, MA, join]
Using our enhanced architecture with range guards, we are able to reply to equality and range queries for a single attribute - and single relation- efficiently. In this section we examine join queries, both without and with range guards.

Let us assume that we have 2 relations, *R* $(DA_1, DA_2, .. DA_k)$ *and S* $(DB_1, DB_2, .. DB_l)$, where DA_i and DB_j, for i $\in$ {1, 2, .. k} and j $\in$ {1, 2, .. l} are the attribute domains of the relations. Both R and S will be represented in the proposed architecture with the approach presented before.

The join query type can be expressed in SQL as:

```
SELECT *
FROM R, S
WHERE R.a = S.b
```

where R.a and S.b are attributes of relations R and S respectively [2].

Note that our architecture ensures that nearby values of attributes are hashed into the same (or near-by) peers, if the domains of the attributes are the same. Specifically, from the definition of the order-preserving hash function h, the following holds, if the domains of R.a and S.b are the same set:

$$R.a = S.b \Rightarrow h(R.a) = h(S.b) \Rightarrow succ(h(R.a)) = succ(h(S.b)) \quad (4)$$

This means that we do not have to transfer tuples from any of the two relations between different peers to compare them, since those tuples from R that may join with tuples from S are stored at the same peer. Therefore, the join is performed at each peer locally, and results are transferred to the requestor. This is an implementation of a hash join method ([18]). However, we should ask all peers in the network in order to find all solutions, which implies a cost of O(N) routing hops resulting in very high network bandwidth requirements and peer node processing loads.

Thus, this approach is not efficient, but significant improvements could be achieved by employing the following optimization techniques (they reduce latency but not the number of hops):

- At each peer, locally scan only those tuples of a relation that their join attributes hash at that peer. (Remember that each peer stores many tuples after hashing each one of the k attributes of the relation). This would reduce time by k, since there are k +1 replicas of all tuples in the whole network, and we scan only one copy of each.).
- Use pipelining to pass partial results to the requestor (users are impatient and we do not have to wait for all results).

However, join processing is still inefficient, since we still have to multicast the query to almost all peers in the network. Therefore, we need to improve join performance. To this end, we employ range guards, as presented earlier.

[2] We examine only equijoins, although nonequijoins would be similarly implemented, since the hashing functions we use maintain the correct ordering of the tuples.

Join Queries using Range Guards

For the values of each join attribute and for all relations, which most possibly will participate in a join operation, we create range guards, as defined earlier. Therefore, a RG stores all tuples whose join attribute values fall in the corresponding range.

The join method in this case is based on *hash-partitioned join* ([18]). According to this method, a hash function - also referred as *split* - is used to partition the tuples in each relation into a fixed number of disjoint sets. The sets are such that a tuple hashes into a given set if the hashed value of its join attributes falls in the range of values for that set. Then, the tuples in the first set of one relation can match only with the tuples in the first set of the second relation. Thus, the processing of different pairs of corresponding partitions from the two sets are independent of other sets.

We split the domain of the join attributes into *l* partitions and assign each partition to one of the l RGs. Therefore, the query is sent to only those *l* RGs (not to all peers in the network, as before). This means that, in case *l*=O(log(N), or *l* = O($\sqrt{N}$) *we* would have similar performance improvements for the processing of joins to those discussed for the processing of range queries. Tuples in partition pairs from both relations may be joined by a *simple hash join*, or any other join method.

Alternative Architecture for Multi-Attribute Query Processing

There is often the case when a few specific attributes are used together in queries. In this case, we could alternatively, use hashing on a combination of these (two or more) attributes.

For example, given a query such as :

```
SELECT attributes
FROM R
WHERE A = Avalue AND B = Bvalue AND C = Cvalue
```

We proceed as follows: We place tuples on the Chord ring as before, except now we hash once only using the tuple's attribute values on A, B, and C, instead of placing the tuple each time at a different Chord node, using a hash on A, then on B, and finally on C.

This would enable faster query processing, since with a single Chord lookup we would reach the peer storing only tuples of interest. However, extensions are required for this approach to enable efficient answers to range queries (such as to define an order on the attribute's vectors, and then an order-preserving hash function over more than one attribute). A special case would be hashing over all k attributes of a tuple (when looking for specific items).

5.2 Non-Rudimentary Query Types

Query Type [SR, MA, =]
This query type is handled similar with the query type [SR, SA, =], except that the lookup algorithm runs for more than once, one for each of the equality-conditions, to find the peers where the requested tuples can be found.

Specifically, hashing is performed for all given values of all attributes to find the corresponding number of peers, where the requested tuples are stored. All matched tuples are sent back to the requestor.

A number of optimizations can be performed in this algorithm, borrowed from the traditional methods in distributed query processing ([19]), in addition to preprocessing of the conditions (i.e. AND, OR, etc) which may significantly improve efficiency.

Query Type [SR, MA, <>]
Similarly with above, we run the rudimentary query type [SR, SA, <>] more than once, one for each of the range-conditions, to find the peers where the requested tuples can be found. Similar techniques to handle the conditions are applied here.

Query Type [MR, MA, =]
We have described above how our enhanced architecture is able to handle more than one relation. This query type is handled similarly with [SR, MA, =], and runs more than once, one for each of the relations. A special case of this query type may be join (i.e. when equality involves attributes of different relations).

Query Type [MR, MA, <>]
As above, the query type [SR, MA, <>] runs more than once.

Query Type [MR, MA, =, sf]
Special cases of this query type, such as the ones involving grouping and ordering, can be handled easily, by using the query types [SR, SA, =] and [SR, SA, <>] and calculating special functions over selected tuples.

Moreover, since tuples are stored in the network in an ordered form, the calculation of a number of functions, such as min, max, average, etc. can be done using similar techniques from ordered lists in combination with the function succ(n). However, with respect to the sum operation, we need to access all peers. Therefore, in that case we can employ range guards to reduce routing hops.

6 Comparison with related work

The work presented here is a straightforward approach to develop a unifying framework to enable structured (Chord-based) P2P networks to support complex query processing. To our knowledge, there is no reported work, which provides such a framework. Instead, solutions have been contributed which focus on supporting one specific query type, such as range, or join/aggregation queries.

The approach proposed in Gupta et. al [9] assumes that users ask broad queries and are not interested for exact answers, and provides thus only approximate answers for range queries, by looking for data in peers that store similar ranges with high probability (i.e. based on similarity). In their experimental results, they report a 50% success in matching partitions with similarity between 0.9 and 1.0, using min-wise independent permutations, and "good" matches for only about 35% of the queries, when using approximate min-wise independent permutations for faster results. Additionally, the min-wise independent permutations answer only 30% of the queries completely, whereas the approximate min-wise independent permutations provide answers that are only 35% complete.

Sahin et al. [11] propose a solution that uses a method to process range queries over the CAN DHT system with d=2. The proposed technique ensures that a range lookup will always yield a range partition that is a superset of the query range, if it exists. However, their approach is not efficient, when compared with solutions based on Chord since for d=2, CAN lookups require O($2N^{1/2}$) hops, significantly less efficient than lookups in Chord which require O(logN) hops.

In Andrzejak and Xu ([12]), another proposal is presented to support range queries. They deal with ineffiency and load communication overhead in querying and updating by proposing directed controlled flooding based on the proximity preservation properties of the Hilbert function. However, their solution is also based on CAN and thus has the abovementioned drawbacks associated with it.

With respect to multi-attribute queries, there is no specific work addressing their efficient processing. Researchers propose applying the same technique they presented for a single attribute, repeatedly for each attribute in the query. We have moved forward by proposing techniques to reduce data storage overheads (by storing pointers to data tuples), and/or storing tuples using indices based on more than one attribute.

With respect to join and other complex operations (such as group by), there is only one work (to our knowledge) that supports them in P2P systems in Harren et. al [15]. However, their approach does not support range queries. Furthermore, employing multicasting to perform joins is not efficient, since all peers must be involved in performing a join, which may create scalability problems such as those found in Gnutella [1] and Freenet [20]. Our proposal involving range guards in conjunction with an order preserving hash function may help in avoiding these costs, by executing joins at only select subsets of the peers.

7 Concluding Remarks

With this work we make a first step towards developing a unifying framework for complex query processing in peer-to-peer data networks. Such a framework is very much lacking: related work has so far focused on supporting only a single query type (typically, range queries or join/aggregate queries). To our knowledge our proposal is the only one that addresses range queries, multi-attribute queries, and queries involving joins.

Our proposal supporting range queries enjoys advantages compared to related work: we provide exact answers (unlike some related work) and do so efficiently since our solution is developed over Chord that processes routing operations more efficiently than CAN, which is employed in competing solutions. We have also proposed more efficient methods to process multi-attribute queries. Finally, we have proposed the notion of range guards, which can significantly improve the performance of range and join queries.

A large number of issues remain open; here we name only a few. The efficiency of query processing depends on the avoidance of formation of data hotspots, routing hotspots, which are currently a problem for all known approaches. With respect especially to range guards, currently we are developing methods to identify them efficiently. Furthermore, when all peers are not aware of a dynamically changing relational schema over which they can pose queries presents several formidable challenges. Dealing efficiently with these issues will give P2P query processing a big boost.

References

1. Gnutella: http://gnutella.wego.com
2. Wilcox B. - Hearn O.: Experiences Deploying a Large-Scale Emergent Network. In 1st International Workshop on Peer-to-Peer Systems, IPTPS'02 (2002)
3. Stoica I., Morris R., Karger D., Kaashoek K. F., Balakrishnan H.: 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. ACM Press (2001) 149-160
4. Ratnasamy S., Francis P., Handley M., Karp R., Shenker S.: A scalable Content-Addressable Network. ACM SIGCOMM '01 (2001).
5. Rowstron A., Druschel P.: Pastry: Scalable, decentralized object location and routing for larg-scale peer-to-peer systems. In Middleware, vol. 2218 of Lecture Notes in Computer Science. Springer (2001) 329-350
6. Zhao Y. B., Kubiatowitcz J., Joseph A.: Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, University of California at Berkley, Computer Science Department (2001)
7. Aberer K.: P-Grid: A self-organizing access structure for P2P information systems. In Proc. of the 6th International Conference on Cooperative Information Systems (CoopIS 2001), Trento, Italy (2001)
8. Gribble S., Halevy A., Ives Z., Rodrig M, Suciu D.: What Can Databases Do for Peer-to-Peer? In Proc. of the WebDB Workshop on Databases and the Web (2001)
9. Gupta A., Agrawal D., Abbadi A. E.: Approximate Range Selection Queries in Peer-to-Peer Systems. In Proc. of the 2003 CIDR Conference (2003)
10. Broder A., Charikar M., Frieze A., Mitzenmacher M.: Min-wise independent permutations (extended abstract). In Proc. of the thirtieth annual ACM symposium on Theory of computing. ACM Press (1998) 327-336
11. Sahin O. D., Gupta A., Agrawal D., Abbadi A. E.: Query Processing Over Peer-to-Peer Data Sharing Systems. Technical Report UCSB/CSD-2002-28, University of California at Santa Barbara (2002)
12. Andrzejak A., Xu Z.: Scalable, Efficient Range Queries for Grid Information Services. In Proc. of the 2nd IEEE International Conference on Peer-to-Peer (2002)

13. Asano T., Ranjan D., Roos T., Welzl E., Widmaier P.: Space Filling Curves and their use in Geometric Data Structures. Theoretical Computer Science, 181, (1997) 3-15
14. Felber P.A., Biersack E. W., Garces-Erice L., Ross K.W., Urvoy-Keller G.: Data Indexing and querying in DHT Peer-to-Peer Networks. Working paper.
15. Harren M., Hellerstein J., Huebch R., Loo B. T., Shenker S., Stoica I.: Complex Queries in DHT-based Peer-to-Peer Networks. In 1st International Workshop on Peer-to-Peer Systems, IPTPS'02 (2002)
16. FIPS180-1. Secure hash Standard. U.S. Department of Commerce/NIST, National Technical Information Service, Springield, VA (1995)
17. Ratnasamy S., Shenker S., Stoica I.: Routing Algorithms for DHTs: Some Open Questions. In 1st International Workshop on Peer-to-Peer Systems, IPTPS'02 (2002)
18. Mishra P., Eich M.: Join Processing in Relational Databases. ACM Computing Surveys, Vol. 24, No. 1 (1992)
19. Kossman D.: The State of the Art in Distributed Query Processing. ACM Computing Surveys (2000)
20. Clarke I., et al: Freenet: A Distributed, Anonymous Information Stoarage and Retrieval System. In Proc. ICSI Workshop on Design Issues in Anonymity and Unobservability (2000)